RAJA Documentation

Release 0.7.0

LLNS

Mar 28, 2019

Contents

RAJA is a software library of C++ abstractions, developed at Lawrence Livermore National Laboratory (LLNL), that enable architecture and programming model portability for high performance computing (HPC) applications. RAJA has two main goals:

1. **To enable application portability with manageable disruption** to algorithms and programming styles.
2. To achieve performance **comparable** to using various programming models (e.g., OpenMP, CUDA, etc.) directly.

RAJA targets portable, parallel loop execution by providing building blocks that extend the generally-accepted *parallel for* idiom.

CHAPTER 1

Background and Motivation

Many HPC applications must achieve high performance across a diverse range of computer architectures including: Mac and Windows laptops, parallel clusters of multicore commodity processors, and large-scale supercomputers with advanced heterogeneous node architectures that combine cutting edge CPU and accelerator (e.g., GPU) processors. Exposing fine-grained parallelism in a portable, high performance manner on varied and potentially disruptive architectures presents significant challenges to developers of large-scale HPC applications. This is especially true at US Department of Energy (DOE) laboratories where, for decades, large investments have been made in highly-scalable MPI-only applications that have been in service over multiple platform generations. Often, maintaining developer and user productivity requires the ability to build single-source application source code bases that can be readily ported to new architectures. RAJA is one C++-based programming model abstraction layer that can help to meet this performance portability challenge.

RAJA provides portable abstractions for simple and complex loops – as well as a variety of loop transformations, reductions, scans, atomic operations, data layouts and views, iteration spaces, etc. Currently available execution patterns supported by different programming model back-ends include: sequential, SIMD, NVIDIA CUDA, OpenMP CPU multi-threading and target offload. Support for Intel Threading Building Blocks (TBB) and AMD ROCm support are under development and considered experimental.

RAJA uses standard C++11 – C++ is the predominant programming language in many LLNL applications. RAJA requirements and design are rooted in a decades of developer experience working on production mesh-based multi-physics applications at LLNL. An important RAJA requirement is that application developers can specialize RAJA concepts for different code implementation patterns and C++ usage, since data structures and algorithms vary widely across applications.

RAJA helps developers insulate application loop kernels from underlying architecture and programming model-specific implementation details. Loop bodies and loop execution are decoupled using C++ lambda expressions (loop bodies) and C++ templates (loop execution methods). This approach promotes the perspective that developers should focus on tuning loop patterns rather than individual loops as much as possible. RAJA makes it relatively straightforward to parameterize an application using execution policy types so that it can be compiled in a specific configuration suitable to a given architecture.

CHAPTER 2

Interacting with the RAJA Team

If you are interested in keeping up with RAJA development and communicating with developers and users, please join our Google Group, or contact the development team via email at `raja-dev@llnl.gov`

If you have questions, find a bug, have ideas about expanding the functionality or applicability, or wish to contribute to RAJA development, please do not hesitate to contact us. We are always interested in improving RAJA and exploring new ways to use it. A brief description of how the RAJA team operates can be found in *Contributing to RAJA*.

CHAPTER 3

What's In This Guide?

If you have some familiarity with RAJA and want to get up and running quickly, check out *Getting Started With RAJA*. This guide contains information about accessing the RAJA code, building it, and basic RAJA usage.

If you are completely new to RAJA, please check out the *RAJA Tutorial*. It contains a discussion of essential C++ concepts and will walk you through a sequence of code examples that show how to use key RAJA features.

See *RAJA Features* for a complete, high-level description of RAJA features (like a reference guide).

Additional information about things to think about when considering whether to use RAJA in an application can be found in *Application Considerations*.

3.1 Getting Started With RAJA

This section will help get you up and running with RAJA quickly.

3.1.1 Requirements

The primary requirement for using RAJA is a C++11 compliant compiler. Accessing various programming model back-ends requires that they be supported by the compiler you chose. Available options and how to enable or disable them are described in *Build Configuration Options*. To build and use RAJA in its simplest form requires:

- C++ compiler with C++11 support
- CMake version 3.8.2 or greater for most back-ends, and version 3.9 or greater for CUDA.

3.1.2 Get the Code

The RAJA project is hosted on GitHub. To get the code, clone the repository into a local working space using the command:

```
$ git clone --recursive https://github.com/LLNL/RAJA.git
```

The `--recursive` argument above is needed to pull in other projects that we use as Git *submodules*. Currently, we have only two:

- BLT build system
- NVIDIA CUB

You probably don't need to know much about either of these projects to start using RAJA. But, if you want to know more, click on the links above.

After running the clone command, a copy of the RAJA repository will reside in a `RAJA` subdirectory where you ran the clone command. You will be on the `develop` branch of RAJA, which is our default branch.

If you forget to pass the `--recursive` argument to the `git clone` command, you can type the following commands after cloning:

```
$ cd RAJA
$ git submodule init
$ git submodule update
```

Either way, the end result is the same and you should be good to go.

Note: Any time you switch branches in RAJA, you need to re-run the 'git submodule update' command to set the Git submodules to what is used by the new branch.

3.1.3 Build and Install

Building and installing RAJA can be very easy or more complicated, depending on which features you want to use and how well you understand how to use your system.

Building RAJA

RAJA uses CMake to configure a build. A basic configuration looks like:

```
$ mkdir build-dir && cd build-dir
$ cmake -DCMAKE_INSTALL_PREFIX=/path/to/install ../
```

Note: Builds must be *out-of-source*. RAJA does not allow building in the source directory, so you must create a build directory.

When you run CMake, it will provide output about the compiler that has been found and which features are discovered. Some RAJA features, like OpenMP support are enabled if they are discovered. For a complete summary of configuration options, please see *Build Configuration Options*.

After CMake successfully completes, you compile RAJA by executing the `make` command in the build directory; i.e.,:

```
$ cd build-dir
$ make
```

If you have access to a multi-core system you can compile in parallel by running `make -j` (to build with all available cores) or `make -j N` to build using N cores.

Note: RAJA is configured to build its unit tests by default. If you do not disable them with the appropriate CMake option, you can run them after the build completes to check if everything compiled properly. The easiest way to do this is to type:

```
$ make test
```

after the build completes.

You can also run individual tests by invoking individual test executables directly. They live in subdirectories in the `test` directory. RAJA tests use the Google Test framework, so you can also run tests via Google Test commands.

Installing RAJA

To install RAJA as a library, run the following command in your build directory:

```
$ make install
```

This will copy RAJA header files to the `include` directory and the RAJA library will be installed in the `lib` directory you specified using the `-DCMAKE_INSTALL_PREFIX` CMake option.

3.1.4 Learning to Use RAJA

If you want to view and run a very simple RAJA example code, a good place to start is located in the file: `RAJA/examples/daxpy.cpp`. After building RAJA with the options you select, the executable for this code will reside in the file: `<build-dir>/examples/bin/daxpy`. Simply type the name of the executable in your build directory to run it; i.e.,:

```
$ ./examples/bin/daxpy
```

The `RAJA/examples` directory also contains many other RAJA example codes you can run and experiment with.

For an overview of all the main RAJA features, see *RAJA Features*. A full tutorial with a variety of examples showing how to use RAJA features can be found in *RAJA Tutorial*.

3.2 RAJA Features

The following sections describe key aspects of the main RAJA features.

3.2.1 Elements of Loop Execution

In this section, we describe the basic elements of RAJA loop kernel execution. `RAJA::forall` and `RAJA::kernel` template methods comprise the RAJA interface for loop execution. `RAJA::forall` methods execute simple loops (e.g., non-nested loops) while `RAJA::kernel` methods support nested loops and other complex loop kernels and transformations.

Note:

- All **forall** and **kernel** methods are in the namespace `RAJA`.

- A `RAJA::forall` loop execution method is a template on an *execution policy* type. A `RAJA::forall` method takes two arguments:
  - an iteration space object, and
  - a lambda expression representing the loop body.
- Each `RAJA::kernel` method is a template on a policy that contains statements with *execution policy* types appropriate for the kernel structure; e.g., an execution policy for each level in a loop nest. A `RAJA::kernel` method takes multiple arguments:
  - a *tuple* of iteration space objects, and
  - one or more lambda expressions representing portions of the loop kernel body.

Various examples showing how to use `RAJA::forall` and `RAJA::kernel` methods may be found in the *RAJA Tutorial*.

For more information on RAJA execution policies and iteration space constructs, see *Policies* and *Indices, Segments, and IndexSets*, respectively.

Simple Loops (RAJA::forall)

As noted earlier, a `RAJA::forall` template executes simple (e.g., non-nested) loops. For example, a C-style loop like:

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

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

may be written in a RAJA form as:

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

RAJA::forall<exec_policy>(iter_space I, [=] (index_type i) {
  c[i] = a[i] + b[i];
});
```

A `RAJA::forall` method is a template on an execution policy type and takes two arguments: an object describing the loop iteration space, such as a RAJA segment or index set, and a lambda expression for the loop body. Applying different loop execution policies enables the loop to run in different ways; e.g., using different programming model back-ends. Different iteration space objects enable the loop iterates to be partitioned, reordered, run in different threads, etc.

Note: Changing loop execution policy types and iteration space constructs enable loops to run in different ways by recompiling the code and without modifying the loop kernel code.

While loop execution using `RAJA::forall` methods is a subset of `RAJA::kernel` functionality, described next, we maintain the `RAJA::forall` interface for simple loop execution because the syntax is simpler and less verbose.

Note: Data arrays in lambda expressions used with RAJA are typically RAJA Views (see *View and Layout*) or bare pointers as shown in the code snippets above. Using something like 'std::vector' is non-portable (won't work in CUDA kernels) and would add excessive overhead for copying data into the lambda data environment.

Complex Loops (RAJA::kernel)

A `RAJA::kernel` template provides ways to compose and execute arbitrary loop nests and other complex kernels. To introduce the RAJA *kernel* interface, consider a (N+1)-level C-style loop nest:

```
for (index_type iN = 0; iN < NN; ++iN) {
  ...
     for (index_type i0 = 0; i0 < N0; ++i0) {s
       \\ inner loop body
     }
}
```

Note that we could write this by nesting `RAJA::forall` statements and it would work, assuming the execution policies were chosen properly:

```
RAJA::forall<exec_policyN>(IN, [=] (index_type iN) {
  ...
     RAJA::forall<exec_policy0>(I0, [=] (index_type i0)) {
       \\ inner loop body
     }
  ...
}
```

However, this approach treats each loop level as an independent entity. This makes it difficult to parallelize the levels in the loop nest together. So it limits the amount of parallelism that can be exposed and the types of parallelism that may be used. For example, if an OpenMP or CUDA parallel execution policy is used on the outermost loop, then all inner loops would be run sequentially in each thread. It also makes it difficult to perform transformations like loop interchange and loop collapse.

The RAJA *kernel* interface facilitates parallel execution and transformations of arbitrary loop nests and other complex loops. It can treat a complex loop structure as a single entity, which simplifies the ability to apply kernel transformations and different parallel execution patterns by changing one execution policy type.

The loop nest may be written in a RAJA kernel form as:

```
using KERNEL_POL =
  RAJA::KernelPolicy< RAJA::statement::For<N, exec_policyN,
                          ...
                            RAJA::statement::For<0, exec_policy0,
                              RAJA::statement::Lambda<0>
                            >
                          ...
                        >
                      >;

RAJA::kernel< KERNEL_POL >(
  RAJA::make_tuple(iter_space IN, ..., iter_space I0),

  [=] (index_type iN, ... , index_type i0) {
     // inner loop body
```

(continues on next page)

(continued from previous page)

```
  }

);
```

A `RAJA::kernel` method takes a `RAJA::KernelPolicy` type template parameter, and a tuple of iteration spaces and a sequence of lambda expressions as arguments.

In the case we discuss here, the execution policy contains a nested sequence of `RAJA::statement::For` statements, one for each level in the loop nest. Each 'For' statement takes three template parameters:

- an integral index parameter that binds it to the item in the iteration space tuple associated with that index,
- an execution policy type for the corresponding loop nest level, and
- an *enclosed statement list* (described in *RAJA Kernel Execution Policies*).

Note: The nesting of `RAJA::statement::For` types is analogous to the nesting of for-statements in the C-style version of the loop nest. A notable syntactic difference is that curly braces are replaced with '<, >' symbols enclosing the template parameter lists.

Here, the innermost type in the kernel policy is a `RAJA::statement::Lambda<0>` type indicating that the first lambda expression (argument zero of the sequence of lambdas passed to the `RAJA::kernel` method) will comprise the inner loop body. We only have one lambda in this example but, in general, we can have any number of lambdas and we can use any subset of them, with `RAJA::statement::Lambda` types placed appropriately in the execution policy, to construct a loop kernel. For example, placing `RAJA::statement::Lambda` types between `RAJA::statement::For` statements enables non-perfectly nested loops.

Each lambda expression passed to a `RAJA::kernel` method **must take an index argument for each iteration space in the tuple**. However, any subset of the arguments may actually be used in each lambda expression.

Note: The loop index arguments for each lambda expression used in a RAJA kernel loop body **must match** the contents of the *iteration space tuple* in number, order, and type. Not all index arguments must be used in each lambda, but they **all must appear** for the RAJA kernel to be well-formed. In particular, your code will not compile if this is not done correctly. If an argument is unused in a lambda expression, you may include its type and omit its name in the argument list to avoid compiler warnings just as one would do for a regular C++ method.

For RAJA nested loops implemented with `RAJA::kernel`, as shown here, the loop nest ordering is determined by the order of the nested policies, starting with the outermost loop and ending with the innermost loop.

Note: The integer value that appears as the first parameter in each `RAJA::statement::For` template indicates which iteration space tuple entry or lambda index argument it corresponds to. **This allows loop nesting order to be changed simply by changing the ordering of the nested policy statements**. This is analogous to changing the order of 'for-loop' statements in C-style nested loop code.

See *Basic RAJA::kernel Variants* for a complete example showing RAJA nested loop functionality and *Nested Loop Interchange* for a detailed example describing nested loop reordering.

A summary of all RAJA execution policies that may be used with `RAJA::forall` or `RAJA::kernel` may be found in *Policies*. Also, a discussion of how to construct `RAJA::KernelPolicy` types and available `RAJA::statement` types can be found in *RAJA Kernel Execution Policies*.

3.2.2 Policies

This section describes various RAJA policies for loop kernel execution, scans, reductions, atomics, etc. Each policy is a type that is passed to a RAJA template method or class to specialize its behavior. Typically, the policy indicates which programming model back-end to use and sometimes provides additional information about the execution pattern, such as number of CUDA threads per threadblock, whether execution is synchronous or asynchronous, etc.

As RAJA functionality is expanded, new policies will be added and some may be redefined and to work in new ways.

Note:

- All RAJA policies are in the namespace `RAJA`.

RAJA Loop/Kernel Execution Policies

The following table summarizes RAJA policies for executing loops and kernels.

The following notes apply to the execution policies described in the table above.

Note: To control the number of threads used by OpenMP policies set the value of the environment variable 'OMP_NUM_THREADS' (which is fixed for duration of run), or call the OpenMP routine 'omp_set_num_threads(nthreads)' (which allows changing number of threads at runtime).

OpenMP Target Policies

- `omp_target_parallel_for_exec<ThreadsPerTeam>` - Execute a loop in parallel using an `omp target parallel for` pragma with given number of threads per team; e.g., if a GPU device is available, this is similar to launching a CUDA kernel with a thread block size of ThreadsPerTeam.

Intel Threading Building Blocks (TBB) Policies

- `tbb_for_exec` - Schedule loop iterations as tasks to execute in parallel using a TBB `parallel_for` method.
- `tbb_for_static<CHUNK_SIZE>` - Schedule loop iterations as tasks to execute in parallel using a TBB `parallel_for` method with a static partitioner using given chunk size.
- `tbb_for_dynamic` - Schedule loop iterations as tasks to execute in parallel using a TBB `parallel_for` method with a dynamic scheduler.

Note: To control the number of TBB worker threads used by these policies: set the value of the environment variable 'TBB_NUM_WORKERS' (which is fixed for duration of run), or create a 'task_scheduler_init' object:

```
tbb::task_scheduler_init TBBinit( nworkers );

// do some parallel work

TBBinit.terminate();
TBBinit.initialize( new_nworkers );

// do some more parallel work
```

This allows changing number of workers at runtime.

Several notable constraints apply to RAJA CUDA thread-direct policies.

Note:

- Repeating thread direct policies with the same thread dimension in perfectly nested loops is not recommended. Your code may do something, but likely will not do what you expect and/or be correct.
- If multiple thread direct policies are used in a kernel (using different thread dimensions), the product of sizes of the corresponding iteration spaces must be $\leq$ 1024. You cannot launch a CUDA kernel with more than 1024 threads per block.
- **Thread-direct policies are recommended only for certain loop patterns, such as tiling.**

Several notes regarding CUDA thread and block loop policies are also good to know.

Note:

- There is no constraint on the product of sizes of the associated loop iteration space.
- These polices allow having a larger number of iterates than threads in the x, y, or z thread dimension.
- **Cuda thread and block loop policies are recommended for most loop patterns.**

RAJA IndexSet Execution Policies

When an IndexSet iteration space is used in RAJA, such as passing an IndexSet to a `RAJA::forall` method, an index set execution policy is required. An index set execution policy is a **two-level policy**: an 'outer' policy for iterating over segments in the index set, and an 'inner' policy used to execute the iterations defined by each segment. An index set execution policy type has the form:

```
RAJA::ExecPolicy< segment_iteration_policy, segment_execution_policy>
```

See *IndexSets* for more information.

In general, any policy that can be used with a `RAJA::forall` method can be used as the segment execution policy. The following policies are available to use for the segment iteration policy:

Execution Policy	Brief description
Serial	
seq_segit	Iterate over index set segments sequentially
OpenMP CPU multithreading	
omp_parallel_segit	Create OpenMP parallel region and iterate over segments in parallel inside it; i.e., apply `omp parallel for` pragma on loop over segments
omp_parallel_for_segit	Same as above
Intel Threading Building Blocks	
tbb_segit	Iterate over index set segments in parallel using a TBB 'parallel_for' method

Parallel Region Policies

The following policies may only be used with the `RAJA::region` method. `RAJA::forall` and `RAJA::kernel` methods may be used within a parallel region created with the `RAJA::region` construct.

- `seq_region` - Create a sequential region (see note below).
- `omp_parallel_region` - Create an OpenMP parallel region.

For example, the following code will execute two consecutive loops in parallel in an OpenMP parallel region without synchronizing threads between them:

```
RAJA::region<RAJA::omp_parallel_region>( [=]() {

  RAJA::forall<RAJA::omp_for_nowait_exec>(
    RAJA::RangeSegment(0, N), [=](int i) {
      // loop body #1
  });

  RAJA::forall<RAJA::omp_for_nowait_exec>(
    RAJA::RangeSegment(0, N), [=](int i) {
      // loop body #2
  });

}); // end omp parallel region
```

Note: The sequential region specialization is essentially a *pass through* operation. It is provided so that if you want to turn off OpenMP in your code, you can simply replace the region policy type and you do not have to change your algorithm source code.

Reduction Policies

Each RAJA reduction object must be defined with a 'reduction policy' type. Reduction policy types are distinct from loop execution policy types. It is important to note the following constraints about RAJA reduction usage:

Note: To guarantee correctness, a **reduction policy must be consistent with the loop execution policy** used. For example, a CUDA reduction policy must be used when the execution policy is a CUDA policy, an OpenMP reduction policy must be used when the execution policy is an OpenMP policy, and so on.

The following table summarizes RAJA reduction policy types:

Reduction Policy	Loop Policies to Use With	Brief description
seq_reduce	seq_exec, loop_exec	Non-parallel (sequential) reduction
omp_reduce	any OpenMP policy	OpenMP parallel reduction
omp_reduce_ordered	any OpenMP policy	OpenMP parallel reduction with result guaranteed to be reproducible
omp_target_reduce	any OpenMP target policy	OpenMP parallel target offload reduction
tbb_reduce	any TBB policy	TBB parallel reduction
cuda_reduce	any CUDA policy	Parallel reduction in a CUDA kernel (device synchronization will occur when reduction value is finalized)
cuda_reduce_atomic	any CUDA policy	Same as above, but reduction may use CUDA atomic operations

Note: RAJA reductions used with SIMD execution policies are not guaranteed to generate correct results at present.

Atomic Policies

Each RAJA atomic operation must be defined with an 'atomic policy' type. Atomic policy types are distinct from loop execution policy types.

Note: An atomic policy type must be consistent with the loop execution policy for the kernel in which the atomic operation is used. The following table summarizes RAJA atomic policies and usage.

Atomic Policy	Loop Policies to Use With	Brief description
seq_atomic	seq_exec, loop_exec	Atomic operation performed in a non-parallel (sequential) kernel
omp_atomic	any OpenMP policy	Atomic operation performed in an OpenMP multithreading or target kernel; i.e., apply `omp atomic` pragma
cuda_atomic	any CUDA policy	Atomic operation performed in a CUDA kernel
builtin_atomic	seq_exec, loop_exec, any OpenMP policy	Compiler *builtin* atomic operation
auto_atomic	seq_exec, loop_exec, any OpenMP policy, any CUDA policy	Atomic operation *compatible* with loop execution policy. See example below.

Here is an example illustrating use of the `auto_atomic` policy:

```
RAJA::forall< RAJA::cuda_exec >(RAJA::RangeSegment seg(0, N),
  [=] RAJA_DEVICE (RAJA::Index_type i) {

  RAJA::atomic::atomicAdd< RAJA::auto_atomic >(&sum, 1);

});
```

In this case, the atomic operation knows that it is used in a CUDA kernel context and the CUDA atomic operation is applied. Similarly, if an OpenMP execution policy was used, the OpenMP version of the atomic operation would be used.

Note:

- There are no RAJA atomic policies for TBB (Intel Threading Building Blocks) execution contexts at present.
- The `builtin_atomic` policy may be preferable to the `omp_atomic` policy in terms of performance.

Local Array Memory Policies

`RAJA::LocalArray` types must use a memory policy indicating where the memory for the local array will live. These policies are described in *Local Array*.

The following memory policies are available to specify memory allocation for `RAJA::LocalArray` objects:

- `RAJA::cpu_tile_mem` - Allocate CPU memory on the stack
- `RAJA::cuda_shared_mem` - Allocate CUDA shared memory
- `RAJA::cuda_thread_mem` - Allocate CUDA thread private memory

RAJA Kernel Execution Policies

RAJA kernel execution policy constructs form a simple domain specific language for composing and transforming complex loops that relies **solely on standard C++11 template support**. RAJA kernel policies are constructed using a combination of *Statements* and *Statement Lists*. A RAJA Statement is an action, such as execute a loop, invoke a lambda, set a thread barrier, etc. A StatementList is an ordered list of Statements that are composed in the order that they appear in the kernel policy to construct a kernel. A Statement may contain an enclosed StatmentList. Thus, a `RAJA::KernelPolicy` type is really just a StatementList.

The main Statement types provided by RAJA are `RAJA::statement::For` and `RAJA::statement::Lambda`, that we have shown above. A 'For' Statement indicates a for-loop structure and takes three template arguments: 'ArgId', 'ExecPolicy', and 'EnclosedStatements'. The ArgID identifies the position of the item it applies to in the iteration space tuple argument to the `RAJA::kernel` method. The ExecPolicy is the RAJA execution policy to use on that loop/iteration space (similar to `RAJA::forall`). EnclosedStatements contain whatever is nested within the template parameter list to form a StatementList, which will be executed for each iteration of the loop. The `RAJA::statement::Lambda<LambdaID>` invokes the lambda corresponding to its position (LambdaID) in the sequence of lambda expressions in the `RAJA::kernel` argument list. For example, a simple sequential for-loop:

```
for (int i = 0; i < N; ++i) {
  // loop body
}
```

can be represented using the RAJA kernel interface as:

```
using KERNEL_POLICY =
  RAJA::KernelPolicy<
    RAJA::statement::For<0, RAJA::seq_exec,
      RAJA::statement::Lambda<0>
    >
  >;

RAJA::kernel<KERNEL_POLICY>(
  RAJA::make_tuple(N_range),
  [=](int i) {
    // loop body
```

(continues on next page)

(continued from previous page)

```
  }
);
```

Note: All `RAJA::forall` functionality can be done using the `RAJA::kernel` interface. We maintain the `RAJA::forall` interface since it is less verbose and thus more convenient for users.

RAJA::kernel Statement Types

The list below summarizes the current collection of statement types that can be used with `RAJA::kernel` and `RAJA::kernel_param`. More detailed explanation along with examples of how they are used can be found in *RAJA Tutorial*.

Note:

- **All of these statement types are in the namespace `RAJA`.**
  - `RAJA::kernel_param` functions similar to `RAJA::kernel` except that its second argument is a *tuple of parameters* used in a kernel for local arrays, thread local variables, tiling information, etc.
- `statement::For< ArgId, ExecPolicy, EnclosedStatements >` abstracts a for-loop associated with kernel iteration space at tuple index 'ArgId', to be run with 'ExecPolicy' execution policy, and containing the 'EnclosedStatements' which are executed for each loop iteration.
- `statement::Lambda< LambdaId >` invokes the lambda expression that appears at position 'LambdaId' in the sequence of lambda arguments.
- `statement::Collapse< ExecPolicy, ArgList<...>, EnclosedStatements >` collapses multiple perfectly nested loops specified by tuple iteration space indices in 'ArgList', using the 'ExecPolicy' execution policy, and places 'EnclosedStatements' inside the collapsed loops which are executed for each iteration. Note that this only works for CPU execution policies (e.g., sequential, OpenMP).It may be available for CUDA in the future if such use cases arise.
- `statement::CudaKernel< EnclosedStatements>` launches 'EnclosedStatements' as a CUDA kernel; e.g., a loop nest where the iteration spaces of each loop level are associated with threads and/or thread blocks as described by the execution policies applied to them.
- `statement::CudaSyncThreads` provides CUDA '__syncthreads' barrier. Note that a similar thread barrier for OpenMP will be added soon.
- `statement::InitLocalMem< MemPolicy, ParamList<...>, EnclosedStatements >` allocates memory for a `RAJA::LocalArray` object used in kernel. The 'ParamList' entries indicate which local array objects in a tuple will be initialized. The 'EnclosedStatements' contain the code in which the local array will be accessed; e.g., initialization operations.
- `statement::Tile< ArgId, TilePolicy, ExecPolicy, EnclosedStatements >` abstracts an outer tiling loop containing an inner for-loop over each tile. The 'ArgId' indicates which entry in the iteration space tuple to which the tiling loop applies and the 'TilePolicy' specifies the tiling pattern to use, including its dimension. The 'ExecPolicy' and 'EnclosedStatements' are similar to what they represent in a `statement::For` type.
- `statement::TileTCount< ArgId, ParamId, TilePolicy, ExecPolicy, EnclosedStatements >` abstracts an outer tiling loop containing an inner for-loop over each tile, **where it is necessary to obtain the tile number in each tile**. The 'ArgId' indicates which entry in the iteration space tuple to which the loop applies and the 'ParamId' indicates the position of the tile number in the

parameter tuple. The 'TilePolicy' specifies the tiling pattern to use, including its dimension. The 'ExecPolicy' and 'EnclosedStatements' are similar to what they represent in a `statement::For` type.

- `statement::tile_fixed<TileSize>` partitions loop iterations into tiles of a fixed size specified by 'TileSize'. This statement type can be used as the 'TilePolicy' template paramter in the Tile statements above.
- `statement::ForICount< ArgId, ParamId, ExecPolicy, EnclosedStatements >` abstracts an inner for-loop within an outer tiling loop **where it is necessary to obtain the local iteration index in each tile**. The 'ArgId' indicates which entry in the iteration space tuple to which the loop applies and the 'ParamId' indicates the position of the tile index parameter in the parameter tuple. The 'ExecPolicy' and 'EnclosedStatements' are similar to what they represent in a `statement::For` type.
- `RAJA::statement::Reduce< ReducePolicy, Operator, ParamId, EnclosedStatements >` reduces a value across threads to a single thread. The 'ReducePolicy' is similar to what it represents for RAJA reduction types. 'ParamId' specifies the position of the reduction value in the parameter tuple passed to the `RAJA::kernel_param` method. 'Operator' is the binary operator used in the reduction; typically, this will be one of the operators that can be used with RAJA scans (see *RAJA Scan Operators*. After the reduction is complete, the 'EnclosedStatements' execute on the thread that received the final reduced value.
- `statement::If< Conditional >` chooses which portions of a policy to run based on run-time evaluation of conditional statement; e.g., true or false, equal to some value, etc.
- `statement::Hyperplane< ArgId, HpExecPolicy, ArgList<...>, ExecPolicy, EnclosedStatements >` provides a hyperplane (or wavefront) iteration pattern over multiple indices. A hyperplane is a set of multi-dimensional index values: i0, i1, ... such that h = i0 + i1 + ... for a given h. Here, 'ArgId' is the position of the loop argument we will iterate on (defines the order of hyperplanes), 'HpExecPolicy' is the execution policy used to iterate over the iteration space specified by ArgId (often sequential), 'ArgList' is a list of other indices that along with ArgId define a hyperplane, and 'ExecPolicy' is the execution policy that applies to the loops in ArgList. Then, for each iteration, everything in the 'EnclosedStatements' is executed.

Examples that show how to use a variety of these statement types can be found in *Complex Loops: Transformations and Advanced RAJA Features*.

3.2.3 Indices, Segments, and IndexSets

Loop variables and their associated iteration spaces are fundamental to writing loop kernels in RAJA. RAJA provides some basic iteration space types that serve as flexible building blocks that can be used to form a variety of loop iteration patterns. These types can be used to define a particular order for loop iterates, aggregate and partition iterates, as well as other configurations. In this section, we introduce RAJA index and iteration space concepts and types.

More examples of RAJA iteration space usage can be found in the *Iteration Spaces: IndexSets and Segments* and *Mesh Vertex Sum Example: Iteration Space Coloring* sections of the tutorial.

Note: All RAJA iteration space types described here are located in the namespace `RAJA`.

Indices

Just like traditional C and C++ for-loops, RAJA uses index variables to identify loop iterates. Any lambda expression that represents all or part of a loop body passed to a `RAJA::forall` or `RAJA::kernel` method will take at least one loop index variable argument. RAJA iteration space types and methods are templates that allow users to use any integral type for an index variable. The index variable type may be explicitly specified by a user. RAJA also

provides a `RAJA::Index_type` type, which is used as a default in some circumstances for convenience by allowing use of a common type alias to typed constructs without explicitly specifying the type. The `RAJA::Index_type` type is an alias to the C++ type 'std::ptrdiff_t', which is appropriate for most compilers to generate useful loop-level optimizations.

Note: Users can change the type of `RAJA::Index_type` by editing the RAJA `RAJA/include/RAJA/util/types.hpp` header file.

Segments

A RAJA **Segment** represents a set of loop indices that one wants to execute as a unit. RAJA provides Segment types for contiguous index ranges, constant (non-unit) stride ranges, and arbitrary lists of indices.

Stride-1 Segments

A `RAJA::TypedRangeSegment` is the fundamental type for representing a stride-1 (i.e., contiguous) range of indices.

Fig. 1: A range segment defines a stride-1 index range [beg, end).

One can create an explicitly-typed range segment or one with the default `RAJA::Index_type` index type. For example,:

```
// A stride-1 index range [beg, end) using type int.
RAJA::TypedRangeSegment<int> int_range(beg, end);

// A stride-1 index range [beg, end) using the RAJA::Index_type default type
RAJA::RangeSegment default_range(beg, end);
```

Note: When using a RAJA range segment, no loop iterations will be run when begin is greater-than-or-equal-to end.

Strided Segments

A `RAJA::TypedRangeStrideSegment` defines a range with a constant stride that is given explicitly stride, including negative stride.

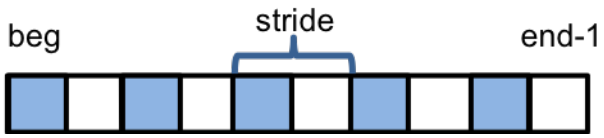

Fig. 2: A range-stride segment defines an index range with arbitrary stride [beg, end, stride).

One can create an explicitly-typed strided range segment or one with the default `RAJA::Index_type` index type. For example,:

```
// A stride-2 index range [beg, end, 2) using type int.
RAJA::TypedRangeStrideSegment<int> stride2_range(beg, end, 2);

// A index range with -1 stride [0, N-1, -1) using the RAJA::Index_type default type
RAJA::RangeStrideSegment neg1_range( N-1, -1, -1);
```

Using a range with a stride of '-1' as above in a RAJA loop traversal template will run the loop indices in reverse order. That is, using 'neg1_range' from above:

```
RAJA::forall< RAJA::seq_exec >( neg1_range, [=] (RAJA::Index_type i) {
  printf("%ld ", i);
} );
```

will print the values:

```
N-1  N-2  N-3 .... 1 0
```

RAJA strided ranges support both positive and negative stride values. The following items are worth noting:

Note: When using a RAJA strided range, no loop iterations will be run under the following conditions:

- Stride > 0 and begin > end
- Stride < 0 and begin < end
- Stride == 0

List Segments

A `RAJA::TypedListSegment` is used to define an arbitrary set of loop indices, akin to an indirection array.

Fig. 3: A list segment defines an arbitrary collection of indices. Here, we have a list segment with 5 irregularly-spaced indices.

A list segment is created by passing an array of integral values to a list segment constructor. For example:

```
// Create a vector holding some integer index values
std::vector<int> idx = {0, 2, 3, 4, 7, 8, 9, 53};

// Create list segment with these loop indices
RAJA::TypedListSegment<int> idx_list( &idx[0], static_cast<int>(idx.size()) );
```

Similar to range segment types, RAJA provides `RAJA::ListSegment`, which is a type alias to `RAJA::TypedListSegment` using `RAJA::Index_type` as the template type parameter.

Segment Types and Iteration

It is worth noting that RAJA segment types model **C++ iterable interfaces**. In particular, each segment type defines three methods:

- begin()
- end()
- size()

and two types:

- iterator (essentially a *random access* iterator type)
- value_type

Thus, any iterable type that defines these methods and types appropriately can be used as a segment with RAJA traversal templates.

IndexSets

A `RAJA::TypedIndexSet` is a container that can hold an arbitrary collection of segment objects of arbitrary type as illustrated in the following figure, where we have two contiguous ranges and an irregularly-spaced list of indices.

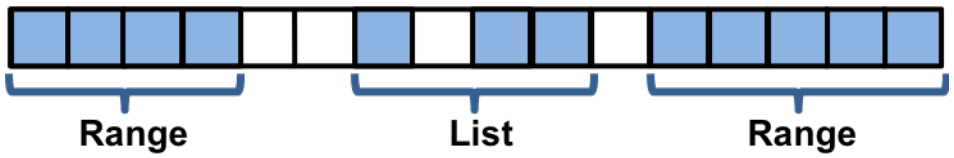

Fig. 4: An index set with 2 range segments and one list segment.

We can create an index set that describes such an iteration space:

```
// Create an index set that can hold range and list segments with the
// default index type
RAJA::TypedIndexSet< RAJA::RangeSegment, RAJA::ListSegment > iset;

// Add two range segments and one list segment to the index set
iset.push_back( RAJA::RangeSegment( ... ) );
iset.push_back( RAJA::ListSegment(...) );
iset.push_back( RAJA::RangeSegment( ... ) );
```

Now that we've created this index set object, we can pass it to any RAJA loop execution template to execute the indices defined by its segments:

```
// Define an index set execution policy type that will iterate over
// its segments in parallel (OpenMP) and execute each segment sequentially
using ISET_EXECPOL = RAJA::ExecPolicy< RAJA::omp_parallel_segit,
                                       RAJA::seq_exec >;

// Run a kernel with iterates defined by the index set
RAJA::forall<ISET_EXECPOL>(iset, [=] (int i) { ... });
```

Note: Iterating over the indices of all segments in a RAJA index set requires a two-level execution policy. The outer level specifies how to iterate over the seqments. The inner level specifies how each segment will execute. See *RAJA IndexSet Execution Policies* for more information about IndexSet execution policies.

In this example, the loop iterations will execute in three chunks defined by the two range segments and one list segment. The segments will be iterated over in parallel using OpenMP, and each segment will execute sequentially.

Note: It is the responsibility of the user to ensure that segments are defined properly when using RAJA index sets. For example, if the same index appears in multiple segments, the corresponding loop iteration will be run multiple times.

3.2.4 View and Layout

Matrix and tensor objects are naturally expressed in scientific computing applications as multi-dimensional arrays. However, for efficiency in C and C++, they are usually allocated as one-dimensional arrays. For example, a matrix A of dimension $N_r \times N_c$ is typically allocated as:

```
double* A = new double [N_r * N_c];
```

Using a one-dimensional array makes it necessary to convert two-dimensional indices (rows and columns of a matrix) to a one-dimensional pointer offset index to access the corresponding array memory location. One could introduce a macro such as:

```
#define A(r, c) A[c + N_c * r]
```

to access a matrix entry in row *r* and column *c*. However, this solution has limitations; e.g., additional macro definitions are needed when adopting a different matrix data layout or when using other matrices. To facilitate multi-dimensional indexing and different indexing layouts, RAJA provides `RAJA::View` and `RAJA::Layout` classes.

RAJA View

A `RAJA::View` object wraps a pointer and enables various indexing schemes based on the definition of a `RAJA::Layout` object. We can create a `RAJA::View` for a matrix with dimensions $N_r \times N_c$ using a RAJA View and a default RAJA two-dimensional Layout as follows:

```
double* A = new double [N_r * N_c];

const int DIM = 2;
RAJA::View<double, RAJA::Layout<DIM> > Aview(A, N_r, N_c);
```

The `RAJA::View` constructor takes a pointer to the matrix data and the extent of each matrix dimension as arguments. The template parameters to the `RAJA::View` type define the pointer type and the Layout type; here, the Layout just defines the number of index dimensions. Using the resulting view object, one may access matrix entries in a row-major fashion (the default RAJA layout) through the View parenthesis operator:

```
// r - row index of a matrix
// c - column index of a matrix
// equivalent to indexing as A[c + r * N_c]
Aview(r, c) = ...;
```

A `RAJA::View` can support any number of index dimensions:

```
const int DIM = n+1;
RAJA::View< double, RAJA::Layout<DIM> > Aview(A, N0, ..., Nn);
```

By default, entries corresponding to the right-most index are contiguous in memory; i.e., unit-stride access. Each other index is offset by the product of the extents of the dimensions to its right. For example, the loop:

```
// iterate over index n and hold all other indices constant
for (int in = 0; in < Nn; ++in) {
  Aview(i0, i1, ..., in) = ...
}
```

accesses array entries with unit stride. The loop:

```
// iterate over index j and hold all other indices constant
for (int j = 0; j < Nj; ++j) {
  Aview(i0, i1, ..., j, ..., iN) = ...
}
```

access array entries with stride $N_n * N_{(n-1)} * \ldots * N_{(j+1)}$.

RAJA Layout

`RAJA::Layout` objects support other indexing patterns with different striding orders, offsets, and permutations. In addition to layouts created using the default Layout constructor, as shown above, RAJA provides other methods to generate layouts for different indexing patterns. We describe these next.

Permuted Layout

The `RAJA::make_permuted_layout` method creates a `RAJA::Layout` object with permuted index strides. That is, the indices with shortest to longest stride are permuted. For example,:

```
std::array< RAJA::idx_t, 3> perm {{1, 2, 0}};
RAJA::Layout<3> layout =
  RAJA::make_permuted_layout( {{5, 7, 11}}, perm );
```

creates a three-dimensional layout with index extents 5, 7, 11 with indices permuted so that the first index (index 0 - extent 5) has unit stride, the third index (index 2 - extent 11) has stride 5, and the second index (index 1 - extent 7) has stride 55 (= 5*11).

Note: If a permuted layout is created with the *identity permutation* (e.g., {0,1,2}, the layout is the same as if it were created by calling the Layout constructor directly with no permutation.

The first argument to `RAJA::make_permuted_layout` is a C++ array whose entries define the extent of each index dimension. **The double braces are required to prevent compilation errors/warnings about issues trying to initialize a sub-object.** The second argument is the striding permutation.

In the next example, we create the same permuted layout, then create a `RAJA::View` with it in a way that tells the View which index has unit stride:

```
const int s0 = 5;  // extent of dimension 0
const int s1 = 7;  // extent of dimension 1
const int s2 = 11; // extent of dimension 2

double* B = new double[s0 * s1 * s2];

std::array< RAJA::idx_t, 3> perm {{1, 2, 0}};
RAJA::Layout<3> layout =
  RAJA::make_permuted_layout( {{s0, s1, s2}}, perm );
```

(continues on next page)

(continued from previous page)

```
// The Layout template parameters are dimension, 'linear index' type,
// and the index with unit stride
RAJA::View<double, RAJA::Layout<3, RAJA::Index_type, 0> > Bview(B, layout);

// Equivalent to indexing as: B[i + j * s0 * s2 + k * s0]
Bview(i, j, k) = ...;
```

Note: Telling a view which index has unit stride makes the multi-dimensional index calculation more efficient by avoiding multiplication by '1' when it is unnecessary. **This must be done so that the layout permutation and unit-stride index specification are the same to prevent incorrect indexing.**

Offset Layout

The `RAJA::make_offset_layout` method creates a `RAJA::Layout` object with offsets applied to the indices. For example,:

```
double* C = new double[11];

RAJA::Layout<1> layout = RAJA::make_offset_layout<1>({{-5}}, {{5}});

RAJA::View<double, RAJA::Layout<1> > Cview(C, layout);
```

creates a one-dimensional view with a layout that allows one to index into it using indices in $[-5, 5]$. In other words, one can use the loop:

```
for (int i = -5; i < 6; ++i) {
  CView(i) = ...;
}
```

to initialize the values of the array. Each 'i' loop index value is converted to array offset access index by subtracting the lower offset to it; i.e., in the loop, each 'i' value has '-5' subtracted from it to properly access the array entry.

The arguments to the `RAJA::make_offset_layout` method are C++ arrays that hold the start and end values of the indices. RAJA offset layouts support any number of dimensions; for example:

```
RAJA::Layout<2> layout = RAJA::make_offset_layout<2>({{-1, -5}}, {{2, 5}});
```

defines a two-dimensional layout that enables one to index into a view using indices $[-1, 2]$ in the first dimension and indices $[-5, 5]$ in the second dimension. As we remarked earlier, double braces are needed to prevent compilation errors/warnings about issues trying to initialize a sub-object.

Permuted Offset Layout

The `RAJA::make_permuted_offset_layout` method creates a `RAJA::Layout` object with permutations and offsets applied to the indices. For example,:

```
std::array< RAJA::idx_t, 2> perm {{1, 0}};
RAJA::Layout<2> layout =
  RAJA::make_permuted_offset_layout<2>( {{-1, -5}}, {{2, 5}}, perm );
```

Here, the two-dimensional index space is $[-1, 2] \times [-5, 5]$, the same as above. However, the index strides are permuted so that the first index (index 0) has unit stride and the second index (index 1) has stride 4, since the first index dimension has length 4.

Complete examples illustrating `RAJA::Layouts` and `RAJA::Views` may be found in the *Stencil Computations (View Offsets)* and *Batched Matrix-Multiply (Permuted Layouts)* tutorial sections.

RAJA Index Mapping

`RAJA::Layout` objects can also be used to map multi-dimensional indices to *linear indices* (i.e., pointer offsets) and vice versa. This section describes basic Layout methods that are useful for converting between such indices. Here, we create a three-dimensional layout with dimension extents 5, 7, and 11 and illustrate mapping between a three-dimensional index space to a one-dimensional linear space:

```
// Create a 5 x 7 x 11 three-dimensional layout object
RAJA::Layout<3> layout(5, 7, 11);

// Map from 3-D index (2, 3, 1) to the linear index
// Note that there is no striding permutation, so rightmost is stride-1
int lin = layout(2, 3, 1); // lin = 188 (= 1 + 3 * 11 + 2 * 11 * 7)

// Map from linear index to 3-D index
int i, j, k;
layout.toIndices(lin, i, j, k); // i,j,k = {2, 3, 1}
```

`RAJA::Layout` also supports *projections*, where one or more dimension extent is zero. In this case, the linear index space is invariant for those multi-dimensional index entries; thus, the 'toIndicies(...)' method will always return zero for each dimension with zero extent. For example:

```
// Create a layout with second dimension extent zero
RAJA::Layout<3> layout(3, 0, 5);

// The second (j) index is projected out
int lin1 = layout(0, 10, 0);   // lin1 = 0
int lin2 = layout(0, 5, 1);    // lin2 = 1

// The inverse mapping always produces a 0 for j
int i,j,k;
layout.toIndices(lin2, i, j, k); // i,j,k = {0, 0, 1}
```

3.2.5 Reduction Operations

RAJA does not provide separate loop execution methods for loops containing reduction operations like some other C++ loop programming abstraction models do. Instead, RAJA provides reduction types that allow users to perform reduction operations in `RAJA::forall` and `RAJA::kernel` methods in a portable, thread-safe manner. Users may use as many reduction objects in a loop kernel as they need. Available RAJA reduction types are described in this section.

A detailed example of RAJA reduction usage can be found in *Reductions*.

Note: All RAJA reduction types are located in the namespace `RAJA`.

Also

Note:

- Each RAJA reduction type is templated on a **reduction policy** and a **reduction value type** for the reduction variable.
- Each RAJA reduction type accepts an **initial reduction value** at construction.
- Each RAJA reduction type has a 'get' method to access its reduced value after kernel execution completes.

Reduction Types

RAJA supports five common reduction types:

- `ReduceSum< reduce_policy, data_type >` - Sum of values.
- `ReduceMin< reduce_policy, data_type >` - Min value.
- `ReduceMax< reduce_policy, data_type >` - Max value.
- `ReduceMinLoc< reduce_policy, data_type >` - Min value and a loop index where the minimum was found.
- `ReduceMaxLoc< reduce_policy, data_type >` - Max value and a loop index where the maximum was found.

Note:

- When `RAJA::ReduceMinLoc` and `RAJA::ReduceMaxLoc` are used in a sequential execution context, the loop index of the min/max is the first index where the min/max occurs.
- When the 'loc' reductions are used in a parallel execution context, the loop index given for the reduction value may be any index where the min or max occurs.

Here is a simple RAJA reduction example that shows how to use a sum reduction type and a min-loc reduction type:

```
const int N = 1000;

//
// Initialize array of length N with all ones. Then, set some other
// values to make the example mildly interesting...
//
int vec[N] = {1};
vec[100] = -10; vec[500] = -10;

// Create sum and min-loc reduction objects with initial values
RAJA::ReduceSum< RAJA::omp_reduce, int > vsum(0);
RAJA::ReduceMinLoc< RAJA::omp_reduce, int > vminloc(100, -1);

RAJA::forall<RAJA::omp_parallel_for_exec>( RAJA::RangeSegment(0, N),
  [=](RAJA::Index_type i) {

  vsum += vec[i];
  vminloc.minloc( vec[i], i );

});
```

(continues on next page)

(continued from previous page)

```
int my_vsum = static_cast<int>(vsum.get());

int my_vmin = static_cast<int>(vminloc.get());
int my_vminloc = static_cast<int>(vminloc.getLoc());
```

The results of these operations will yield the following values:

- my_vsum == 978 (= 998 - 10 - 10)
- my_vmin == -10
- my_vminloc == 100 or 500

Note that the location index for the minimum array value can be one of two values depending on the order of the reduction finalization since the loop is run in parallel.

Reduction Policies

For more information about available RAJA reduction policies and guidance on which to use with RAJA execution policies, please see *Reduction Policies*.

3.2.6 Atomics

RAJA provides portable atomic operations that can be used to update values at arbitrary memory locations while avoiding data races. They are described in this section.

A complete working example code that shows RAJA atomic usage can be found in *Computing a Histogram with Atomic Operations*.

Note:

- All RAJA atomic operations are in the namespace `RAJA::atomic`.

Atomic Operations

RAJA atomic support includes a variety of the most common atomic operations.

Note:

- Each RAJA atomic operation is templated on an *atomic policy*.
- Each method described in the table below returns the value of the potentially modified argument (i.e., *acc) immediately before the atomic operation is applied, in case it is needed by a user.
- Please set the nvcc option "-arch=sm_35" (or greater), when CUDA is enabled. If a CUDA architecture does not support an sm_xy atomic operation, RAJA will generically implement it with atomicCAS.

Arithmetic

- `atomicAdd< atomic_policy >(T* acc, T value)` - Add value to *acc.
- `atomicSub< atomic_policy >(T* acc, T value)` - Subtract value from *acc.

Min/max

- `atomicMin< atomic_policy >(T* acc, T value)` - Set *acc to min of *acc and value.
- `atomicMax< atomic_policy >(T* acc, T value)` - Set *acc to max of *acc and value.

Increment/decrement

- `atomicInc< atomic_policy >(T* acc)` - Add 1 to *acc.
- `atomicDec< atomic_policy >(T* acc)` - Subtract 1 from *acc.
- `atomicInc< atomic_policy >(T* acc, T compare)` - Add 1 to *acc if *acc < compare, else set *acc to zero.
- `atomicDec< atomic_policy >(T* acc, T compare)` - Subtract 1 from *acc if *acc != 0 and *acc <= compare, else set *acc to compare.

Bitwise operations

- `atomicAnd< atomic_policy >(T* acc, T value)` - Bitwise 'and' equivalent: Set *acc to *acc & value. Only works with integral data types.
- `atomicOr< atomic_policy >(T* acc, T value)` - Bitwise 'or' equivalent: Set *acc to *acc | value. Only works with integral data types.
- `atomicXor< atomic_policy >(T* acc, T value)` - Bitwise 'xor' equivalent: Set *acc to *acc ^ value. Only works with integral data types.

Replace

- `atomicExchange< atomic_policy >(T* acc, T value)` - Replace *acc with value.
- `atomicCAS< atomic_policy >(T* acc, Tcompare, T value)` - Compare and swap: Replace *acc with value if and only if *acc is equal to compare.

Here is a simple example that shows how to use an atomic operation to compute an integral sum on a CUDA GPU device:

```
//
// Use CUDA UM to share data pointer with host and device code.
// RAJA mechanics work the same way if device data allocation
// and host-device copies are done with traditional cudaMalloc
// and cudaMemcpy.
//
int* sum = nullptr;
cudaMallocManaged((void **)&sum, sizeof(int));
cudaDeviceSynchronize();
*sum = 0;

RAJA::forall< RAJA::cuda_exec >(RAJA::RangeSegment(0, N),
  [=] RAJA_DEVICE (RAJA::Index_type i) {

  RAJA::atomic::atomicAdd< RAJA::cuda_atomic >(sum, 1);

});
```

After this kernel executes, '*sum' will be equal to 'N'.

AtomicRef

RAJA also provides an atomic interface similar to the C++20 'std::atomic_ref', but which works for arbitrary memory locations. The class `RAJA::atomic::AtomicRef` provides an object-oriented interface to the atomic methods described above. For example, after the following operations:

```
double val = 2.0;
RAJA::atomic::AtomicRef<double,  RAJA::omp_atomic > sum(&val);

sum++;
++sum;
sum += 1.0;
```

the value of 'val' will be 5.

Atomic Policies

For more information about available RAJA atomic policies, please see *Atomic Policies*.

3.2.7 Scans

RAJA provides portable parallel scan operations, which are basic parallel algorithm building blocks. They are described in this section.

A few important notes:

Note:

- All RAJA scan operations are in the namespace `RAJA`.
- Each RAJA scan operation is a template on an *execution policy* parameter. The same policy types used for `RAJA::forall` methods may be used for RAJA scans.
- RAJA scan operations accept an optional *operator* argument so users can perform different types of scan operations. If no operator is given, the default is a 'plus' operation and the result is a **prefix-sum**.

Also:

Note: For scans using the CUDA back-end, RAJA uses the implementations provided by the NVIDIA cub library, which is available in the RAJA source repository as a Git submodule. The CMake variable `CUB_DIR` will be automatically set to the location of the cub library when CUDA is enabled; to use a different version of the cub library, install it and set the `CUB_DIR` variable to the desired location when running CMake.

Please see the *Parallel Scan Operations* tutorial section for usage examples of RAJA scan operations.

Scan Operations

In general, a scan operation takes a sequence of numbers 'x' and a binary associative operator 'op' as input and produces another sequence of numbers 'y' as output. Each element of the output sequence is formed by applying the operator to a subset of the input. Scans come in two flavors: *inclusive* and *exclusive*.

An **inclusive scan** takes the input sequence

$x = \{ x_0, x_1, x_2, \ldots \}$

and calculates the output sequence:

$y = \{ y_0, y_1, y_2, \ldots \}$

using the recursive definition

$y_0 = x_0$

$y_i = y_{i-1} \text{ op } x_i$, for each $i > 0$

An **exclusive scan** is similar, but the output of an exclusive scan is different from the output of an inclusive scan in two ways. First, the first element of the output is the identity of the operator used. Second, the rest of the output sequence is the same as inclusive scan, but shifted one position to the right; i.e.,

$y_0 = \text{op}_{\text{identity}}$

$y_i = y_{i-1} \text{ op } x_{i-1}$, for each $i > 0$

If you would like more information about scan operations, a good overview of what they are and why they are useful can be found in Blelloch Scan Lecture Notes. A nice presentation that describes how parallel scans are implemented is Va Tech Scan Lecture

RAJA Inclusive Scans

RAJA inclusive scan operations look like the following:

- `RAJA::inclusive_scan< exec_policy >(in, in + N, out)`
- `RAJA::inclusive_scan< exec_policy >(in, in + N, out, operator)`

Here, 'in' and 'out' are pointers to arrays of some numeric scalar type whose elements are the input and output sequences of the scan, respectively. The scalar type must be the same for both arrays. The first scan operation above will be a *prefix-sum* since there is no operator argument given; i.e., the output array will contain partial sums of the input array. The second scan will apply the operator that is passed.

RAJA also provides *in-place* scans:

- `RAJA::inclusive_scan_inplace< exec_policy >(in, in + N)`
- `RAJA::inclusive_scan_inplace< exec_policy >(in, in + N, <operator>)`

An in-place scan generates the same output sequence as a non-inplace scan. However, an in-place scan does not take separate input and output arrays and the result of the scan operation will appear *in-place* in the input array.

RAJA Exclusive Scans

Using RAJA exclusive scans is essentially the same as for inclusive scans:

- `RAJA::exclusive_scan< exec_policy >(in, in + N, out)`
- `RAJA::exclusive_scan< exec_policy >(in, in + N, out, operator)`
- `RAJA::exclusive_scan_inplace< exec_policy >(in, in + N)`
- `RAJA::exclusive_scan_inplace< exec_policy >(in, in + N, <operator>)`

RAJA Scan Operators

RAJA provides a variety of operators that can be used to perform different types of scans, such as:

- `RAJA::operators::plus<T>`
- `RAJA::operators::minus<T>`
- `RAJA::operators::multiplies<T>`
- `RAJA::operators::divides<T>`
- `RAJA::operators::minimum<T>`
- `RAJA::operators::maximum<T>`

Note:

- All RAJA scan operators are in the namespace `RAJA::operators`.

Scan Policies

For information about RAJA execution policies to use with scan operations, please see *Policies*.

3.2.8 Local Array

This section introduces RAJA local arrays. A `RAJA::LocalArray` is a multi-dimensional array object whose memory is allocated when a RAJA kernel is executed and only lives within the scope of the kernel execution. To motivate the concept and usage, consider a simple C++ example in which we construct and use two arrays in nested loops:

```
for(int k = 0; k < 7; ++k) { //k loop

 int a_array[7][5];
 int b_array[5];

  for(int j = 0; j < 5; ++j) { //j loop
    a_array[k][j] = 5*k + j;
    b_array[j] = 7*j + k;
  }

  for(int j = 0; j < 5; ++j) { //j loop
    printf("%d %d \n",a_array[k][j], b_array[j]);
  }

}
```

Here, two stack-allocated arrays are defined inside the outer 'k' loop and used in both inner 'j' loops. This loop pattern may be also be expressed using RAJA local arrays in a `RAJA::kernel_param` kernel. We show a RAJA variant below, which matches the implementation above, and then discuss its constituent parts:

```
//
// Define two local arrays
//

using RAJA_a_array = RAJA::LocalArray<int, RAJA::Perm<0, 1>, RAJA::SizeList<5,7> >;
```

(continues on next page)

(continued from previous page)

```
RAJA_a_array kernel_a_array;

using RAJA_b_array = RAJA::LocalArray<int, RAJA::Perm<0>, RAJA::SizeList<5> >;
RAJA_b_array kernel_b_array;


//
// Define the kernel execution policy
//

using POL = RAJA::KernelPolicy<
               RAJA::statement::For<1, RAJA::loop_exec,
                 RAJA::statement::InitLocalMem<RAJA::cpu_tile_mem, RAJA::ParamList<0,␣
↪1>,
                   RAJA::statement::For<0, RAJA::loop_exec,
                     RAJA::statement::Lambda<0>
                   >,
                   RAJA::statement::For<0, RAJA::loop_exec,
                     RAJA::statement::Lambda<1>
                   >
                 >
               >
             >;


//
// Define the kernel
//

RAJA::kernel_param<POL> ( RAJA::make_tuple(RAJA::RangeSegment(0,5),
                                           RAJA::RangeSegment(0,7)),
                          RAJA::make_tuple(kernel_a_array, kernel_b_array),

  [=] (int j, int k, RAJA_a_array& kernel_a_array, RAJA_b_array& kernel_b_array) {
    a_array(k, j) = 5*k + j;
    b_array(j) = 5*k + j;
  },

  [=] (int j, int k, RAJA_a_array& a_array, RAJA_b_array& b_array) {
    printf("%d %d \n", kernel_a_array(k, j), kernel_b_array(j));
  }

);
```

The RAJA version defines two `RAJA::LocalArray` types, one two-dimensional and one one-dimensional and creates an instance of each type. The template arguments for the `RAJA::LocalArray` types are:

- Array data type
- Index permutation (see *View and Layout* for more on layouts and permutations)
- Array dimensions

Note: `RAJA::LocalArray` types support arbitrary dimensions and sizes.

The kernel policy is a two-level nested loop policy (see loop_elements-kernel-label‘ for more information) with a statement type `RAJA::statement::InitLocalMem` inserted between the nested for-loops which allocates the

memory for the local arrays when the kernel executes. The `InitLocalMem` statement type uses a 'CPU tile' memory type, for the two entries '0' and '1' in the kernel parameter tuple (second argument to `RAJA::kernel_param`). Then, the inner initialization loop and inner print loops are run with the respective lambda bodies defined in the kernel.

Memory Policies

`RAJA::LocalArray` supports CPU stack-allocated memory and CUDA GPU shared memory and thread private memory. See *Local Array Memory Policies* for a discussion of available memory policies.

3.2.9 Loop Tiling

In this section, we discuss RAJA statements that can be used to tile nested for-loops. Typical loop tiling involves partitioning an iteration space into a collection of "tiles" and then iterating over tiles in outer loops and entries within each tile in inner loops. Many scientific computing algorithms can benefit from loop tiling due to more efficient cache usage and other considerations.

For example, an operation performed using a for-loop with a range of [0, 10):

```
for (int i=0; i<10; ++i) {
  // loop body using index 'i'
}
```

May be expressed as a loop nest that iterates over five tiles of size two:

```
int numTiles = 5;
int tileDim  = 2;
for (int t=0; t<numTiles; ++t) {
  for (int j=0; j<tileDim; ++j) {
    int i = j + tileDim*t; // Calculate global index 'i'
    // loop body using index 'i'
  }
}
```

Next, we show how this tiled loop can be represented using RAJA. Then, we present variations on it that illustrate the usage of different RAJA kernel statement types.

```
using KERNEL_EXEC_POL =
  RAJA::KernelPolicy<
    RAJA::statement::Tile<0, RAJA::statement::tile_fixed<2>, RAJA::seq_exec,
      RAJA::statement::For<0, RAJA::seq_exec,
        RAJA::statement::Lambda<0>
      >
    >
  >;

RAJA::kernel<KERNEL_EXEC_POL>(RAJA::make_tuple(RAJA::RangeSegment(0,10)),
  [=] (int i) {
  // loop body using index 'i'
});
```

In RAJA, the simplest way to tile an iteration space is to use RAJA `statement::Tile` and `statement::For` statement types. A `statement::Tile` type is similar to a `statement::For` type, but takes a tile size as the second template argument. The Tile statement generates the outer loop over tiles and the For statement iterates over each tile. Nested together, as in the example, these statements will pass the global index 'i' to the loop body in the lambda expression as in the non-tiled version above.

Note: When using `statement::Tile` and `statement::For` types together to define a tiled loop structure, the integer passed as the first template argument to each statement type must be the same. This indicates that they both apply to the same item in the iteration space tuple passed to the `RAJA::kernel` methods.

RAJA also provides alternative Tile and For statements that provide the tile number and local tile index, if needed inside the kernel body, as shown below:

```
using KERNEL_EXEC_POL2 =
  RAJA::KernelPolicy<
    RAJA::statement::TileTCount<0, RAJA::statement::Param<0>,
                                RAJA::statement::tile_fixed<2>, RAJA::seq_exec,
      RAJA::statement::ForICount<0, RAJA::statement::Param<1>,
                                 RAJA::seq_exec,
        RAJA::statement::Lambda<0>
      >
    >
  >;


RAJA::kernel_param<KERNEL_EXEC_POL2>(RAJA::make_tuple(RAJA::RangeSegment(0,10)),
                                     RAJA::make_tuple((int)0, (int)0),
  [=](int i, int t, int j) {

    // i - global index
    // t - tile number
    // j - index within tile
    // Then, i = j + 2*t (2 is tile size)

 });
```

The `statement::TileTCount` type allows the tile number to be accessed as a parameter and the `statement::ForICount` type allows the local tile loop index to be accessed. These values are specified in the tuple, which is the second argument passed to the `RAJA::kernel_param` method above. The `statement::Param<#>` type appearing as the second template parameter for each statement type indicates which parameter tuple entry the tile number or local tile loop index is passed to the lambda, and in what order. Here, the tile number is the second lambda argument (tuple parameter '0') and the local tile loop index is the third lambda argument (tuple parameter '1').

Note: The global loop indices always appear as the first lambda expression arguments. Then, the parameter tuples, identified by the integers in the `Param` statement types given for the loop statement types follow.

3.3 Application Considerations

Warning: Comming soon!! Stay tuned.

3.4 RAJA Tutorial

This RAJA tutorial introduces RAJA concepts and capabilities via a sequence of examples of increasing complexity. Complete working codes for the examples are located in the `RAJA``examples` directory.

To understand the discussion and code examples, a working knowledge of C++ templates and lambda expressions is required. So, before we begin, we provide a bit of background discussion of basic aspects of C++ lambda expressions, which are essential to using RAJA successfully.

To understand the GPU examples (e.g., CUDA), It is also important to know the difference between CPU (host) and GPU (device) memory allocations and how transfers between those memory spaces work. For a detailed discussion, see Device Memory.

RAJA does not provide a memory model by design. Thus, users are responsible for ensuring that data is properly allocated and initialized on a GPU device when running GPU code. This can be done using explicit host and device allocation and copying between host and device memory spaces or via CUDA unified memory (UM), if available. RAJA developers also support a library called CHAI which complements RAJA by providing a simple alternative to manual CUDA calls or UM. For more information, see *Plugins*.

3.4.1 A Little C++ Lambda Background

RAJA is used most easily and effectively by employing C++ lambda expressions for the bodies of loop kernels. Alternatively, C++ functors can be used, but we don't recommend them as they require more source code and have a potentially significant negative impact on source code readability.

Elements of C++ Lambda Expressions

Here, we provide a brief description of the basic elements of C++ lambda expressions. A more technical and detailed discussion is available here: Lambda Functions in C++11 - the Definitive Guide

Lambda expressions were introduced in C++ 11 to provide a lexical-scoped name binding; that is, a *closure* that stores a function with a data environment. In particular, a lambda expression can *capture* variables from an enclosing scope for use within the local scope of the function expression.

A C++ lambda expression has the following form:

```
[capture list] (parameter list) {function body}
```

The `capture list` specifies how variables outside the lambda scope are pulled into the lambda data environment. The `parameter list` defines arguments passed to the lambda function body – for the most part, lambda arguments are just like arguments to a standard C++ method. Variables in the capture list are initialized when the lambda expression is created, while those in the parameter list are set when the lambda expression is called. The body of a lambda expression is similar to the body of an ordinary C++ method. RAJA templates, such as `RAJA::forall` and `RAJA::kernel` pass arguments to lambdas based on usage and context; typically, these are loop indices.

A C++ lambda expression can capture variables in the capture list by value or by reference. This is similar to how arguments to C++ methods are passed; e.g., pass-by-reference or pass-by-value. However, there are some subtle differences between lambda variable capture rules and those for ordinary methods. Variables mentioned in the capture list with no extra symbols are captured by value. Capture-by-reference is accomplished by using the reference symbol '&' before the variable name; for example:

```
int x;
int y = 100;
[&x, &y](){ x = y; };
```

generates a lambda expression that captures both 'x' and 'y' by reference and assigns the value of 'y' to 'x' when called. The same outcome would be achieved by writing:

```
[&](){ x = y; };   // capture all lambda arguments by reference...
```

or:

```
[=, &x](){ x = y; };  // capture 'x' by reference and 'y' by value...
```

Note that the following two attempts will generate compilation errors:

```
[=](){ x = y; };      // capture all lambda arguments by value...
[x, &y](){ x = y; };  // capture 'x' by value and 'y' by reference...
```

Specifically, it is illegal to assign a value to a variable 'x' that is captured by value since it is *read-only*.

Notes About C++ Lambdas

There are several issues to note about C++ lambda expressions; in particular, with respect to RAJA usage. We describe them here.

- **Prefer by-value lambda capture.**

  We recommended *capture by-value* for all lambda loop bodies passed to RAJA execution methods. To execute a RAJA loop on a non-CPU device, such as a GPU, all variables accessed in the loop body must be passed into the GPU device data environment. Using capture by-value for all RAJA-based lambda usage will allow your code to be portable for either CPU or GPU execution. In addition, the read-only nature of variables captured by-value can help avoid incorrect CPU code since the compiler will report incorrect usage.

- **Must use 'device' annotation for CUDA device execution.**

  Any lambda passed to a CUDA execution context (or function called from a CUDA device kernel, for that matter) must be decorated with the `__device__` annotation; for example:

  ```
  RAJA::forall<RAJA::cuda_exec>( range, [=] __device__ (int i) { ... } );
  ```

  Without this, the code will not compile and generate compiler errors indicating that a 'host' lambda cannot be called from 'device' code.

  RAJA provides the macro `RAJA_DEVICE` that can be used to help switch between host-only or device-only CUDA compilation.

- **Avoid 'host-device' annotation on a lambda that will run in host code.**

  RAJA provides the macro `RAJA_HOST_DEVICE` to support the dual CUDA annotation `__host__ __device__`. This makes a lambda or function callable from CPU or CUDA device code. However, when CPU performance is important, **the host-device annotation should not be used on a lambda that is used in a host (i.e., CPU) execution context**. Unfortunately, a loop kernel containing a lambda annotated in this way will run noticeably slower on a CPU than the same lambda with no annotation.

- **Cannot use 'break' and 'continue' statements in a lambda.**

  In this regard, a lambda expression is similar to a function. So, if you have loops in your code with these statements, they should be rewritten.

- **Global variables are not captured in a lambda.**

  This fact is due to the C++ standard. If you need (read-only) access to a global variable inside a lambda expression, one solution is to make a local reference to it; for example:

  ```
  double& ref_to_global_val = global_val;

  RAJA::forall<RAJA::cuda_exec>( range, [=] __device__ (int i) {
    // use ref_to_global_val
  } );
  ```

- **Local stack arrays are not captured by CUDA device lambdas.**

  Although this is inconsistent with the C++ standard, attempting to access elements in a local stack array in a CUDA device lambda will generate a compilation error. One solution to this problem is to wrap the array in a struct; for example:

```
struct array_wrapper {
  int[4] array;
} bounds;

bounds.array = { 0, 1, 8, 9 };

RAJA::forall<RAJA::cuda_exec>(range, [=] __device__ (int i) {
  // access entries of bounds.array
} );
```

3.4.2 RAJA Examples

The remainder of this tutorial illustrates how to use RAJA features using various working code examples that are located in the `RAJA/examples` directory. Additional information about the RAJA features used can be found in *RAJA Features*.

The examples demonstrate CPU execution (sequential, SIMD, OpenMP multi-threading) and CUDA GPU execution. Examples that show how to use RAJA with other parallel programming model back-ends that are in development will appear when we feel RAJA support for them is sufficiently complete and robust. For adventurous users who wish to try experimental features, usage is similar to what is shown in the examples here.

All RAJA programming model support features are enabled via CMake options, which are described in *Build Configuration Options*.

For the purposes of discussion of each example, we assume that any and all data used has been properly allocated and initialized. This is done in the example code files, but is not discussed further here.

3.4.3 Simple Loops and Basic RAJA Features

The examples in this section illustrate how to use `RAJA::forall` methods to execute simple loop kernels; i.e., non-nested loops. It also describes iteration spaces, reductions, atomic operations, and scans.

Vector Addition (Basic Loop Execution)

Key RAJA features shown in this example:

- `RAJA::forall` loop execution template
- `RAJA::RangeSegment` iteration space construct
- RAJA execution policies

In the example, we add two vectors ‘a’ and ‘b’ of length N and store the result in vector ‘c’. A simple C-style loop that does this is:

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

RAJA Variants

The RAJA variants of the vector addition operation illustrate how the same kernel can be run with a variety of different programming model back-ends by simply swapping out the execution policy. This can be done by defining type aliases in a header file so that execution policy types can be easily switched, and the code can be compiled to run differently, without changing the loop kernel code. In the example code, we make all execution policy types explicit for clarity.

For the RAJA variants, we replace the C-style for-loop with a call to the `RAJA::forall` loop execution template method. The method takes an iteration space and the vector addition loop body as a C++ lambda expression. We pass a `RAJA::RangeSegment` object, which describes a contiguous sequence of integral values [0, N) for the iteration space (for more information about RAJA loop indexing concepts, see *Indices, Segments, and IndexSets*). The loop execution template method requires an execution policy template type that specifies how the loop is to run (for more information about RAJA execution policies, see *Policies*).

For the RAJA sequential variant, we use the `RAJA::seq_exec` execution policy type:

```
RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});
```

The RAJA sequential execution policy enforces strictly sequential execution; in particular, no SIMD vectorization instructions or other substantial optimizations will be generated by the compiler. To attempt to force the compiler to generate SIMD vector instructions, we would use the RAJA SIMD execution policy:

```
RAJA::simd_exec
```

Alternatively, RAJA provides a *loop execution* policy:

```
RAJA::loop_exec
```

This policy allows the compiler to generate optimizations, such as SIMD if compiler heuristics suggest that it is safe to do so and potentially beneficial for performance, but the optimizations are not forced.

To run the kernel with OpenMP multi-threaded parallelism on a CPU, we use the `RAJA::omp_parallel_for_exec` execution policy:

```
RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});
```

This will distribute the loop iterations across CPU threads and run the loop over threads in parallel.

Finally, to run the kernel on a CUDA GPU device, we use the `RAJA::cuda_exec` policy:

```
RAJA::forall<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(RAJA::RangeSegment(0, N),
  [=] RAJA_DEVICE (int i) {
  c[i] = a[i] + b[i];
});
```

Note that the CUDA execution policy type accepts a template argument `CUDA_BLOCK_SIZE`, which specifies that each CUDA thread block launched to execute the kernel will have the given number threads in the block. The thread block size parameter is optional; if not provided, the RAJA policy provides a default of 256, which is a reasonable choice for most cases.

Since the lambda defining the loop body will be passed to a device kernel, it must be decorated with the `__device__` attribute when it is defined. This can be done directly or by using the `RAJA_DEVICE` macro.

The file `RAJA/examples/tut_add-vectors.cpp` contains the complete working example code.

Vector Dot Product (Sum Reduction)

Key RAJA features shown in this example:

- `RAJA::forall` loop execution template
- `RAJA::RangeSegment` iteration space construct
- RAJA execution policies
- `RAJA::ReduceSum` sum reduction template
- RAJA reduction policies

In the example, we compute a vector dot product, 'dot = (a,b)', where 'a' and 'b' are two vectors length N and 'dot' is a scalar. Typical C-style code to compute the dot product and print its value afterward is:

```
double dot = 0.0;

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

std::cout << "\t (a, b) = " << dot << std::endl;
```

Note that this operation performs a *reduction*, a computational pattern that produces a single result from a set of values. Reductions present a variety of issues that must be addressed to operate properly in parallel.

RAJA Variants

Different programming models support parallel reduction operations differently. Some models, such as CUDA, do not provide support for reductions at all and so such operations must be explicitly coded by users. It can be challenging to generate a correct and high performance implementation. RAJA provides portable reduction types that make it easy to perform reduction operations in loop kernels. The RAJA variants of the dot product computation show how to use the `RAJA::ReduceSum` sum reduction template type. RAJA provides other reduction types and also allows multiple reduction operations to be performed in a single kernel along with other computation. Please see *Reductions* for an example that does this.

Each RAJA reduction type takes a *reduce policy* template argument, which **must be compatible with the execution policy** applied to the kernel in which the reduction is used. Here is the RAJA sequential variant of the dot product computation:

```
RAJA::ReduceSum<RAJA::seq_reduce, double> seqdot(0.0);

RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  seqdot += a[i] * b[i];
});

dot = seqdot.get();
std::cout << "\t (a, b) = " << dot << std::endl;
```

The sum reduction object is defined by specifying the reduction policy `RAJA::seq_reduce`, which matches the loop execution policy, and a reduction value type (i.e., 'double'). An initial value of zero for the sum is passed to the reduction object constructor. After the kernel executes, we use the 'get' method to retrieve the reduced value.

The OpenMP multi-threaded variant of the loop is implemented similarly:

```
RAJA::ReduceSum<RAJA::omp_reduce, double> ompdot(0.0);

RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  ompdot += a[i] * b[i];
});

dot = ompdot.get();
std::cout << "\t (a, b) = " << dot << std::endl;
```

Here, we use the `RAJA::omp_reduce` reduce policy to match the OpenMP loop execution policy.

Finally, the RAJA CUDA variant is achieved by using appropriate loop execution and reduction policies:

```
RAJA::ReduceSum<RAJA::cuda_reduce, double> cudot(0.0);

RAJA::forall<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(RAJA::RangeSegment(0, N),
  [=] RAJA_DEVICE (int i) {
  cudot += a[i] * b[i];
});

dot = cudot.get();
std::cout << "\t (a, b) = " << dot << std::endl;
```

Here, the CUDA reduce policy `RAJA::cuda_reduce` matches the CUDA loop execution policy. Note that the CUDA thread block size is not specified in the reduce policy as it will use the same value as the loop execution policy.

It is worth noting how similar the code looks for each of these variants. The loop body is identical for each and only the loop execution policy and reduce policy types change.

The file `RAJA/examples/tut_dot-product.cpp` contains the complete working example code.

Iteration Spaces: IndexSets and Segments

Key RAJA features shown in this example:

- `RAJA::forall` loop execution template
- `RAJA::RangeSegment` (i.e., `RAJA::TypedRangeSegment`) iteration space construct
- `RAJA::TypedListSegment` iteration space construct
- `RAJA::IndexSet` iteration construct and associated execution policies

The example uses a simple daxpy kernel and its usage of RAJA is similar to previous simple loop examples. The example focuses on how to use RAJA index sets and iteration space segments, such as index ranges and lists of indices. These features are important for applications and algorithms that use indirection arrays for irregular array accesses. Combining different segment types, such as ranges and lists in an index set allows a user to launch different iteration patterns in a single loop execution construct (i.e., one kernel). This is something that is not supported by other programming models and abstractions and is unique to RAJA. Applying these concepts judiciously can increase performance by allowing compilers to optimize for specific segment types (e.g., SIMD for range segments) while providing the flexibility of indirection arrays for general indexing patterns.

Note: For the following examples, it is useful to remember that all RAJA segment types are templates, where the type of the index value is the template argument. So for example, the basic RAJA range segment type is `RAJA::TypedRangeSegment<T>`. The type `RAJA::RangeSegment` used here (for convenience) is a type alias for `RAJA::TypedRangeSegment<RAJA::Index_type>`, where the template parameter is a default index type that RAJA defines.

For a summary discussion of RAJA segment and index set concepts, please see *Indices, Segments, and IndexSets*.

RAJA Segments

In previous examples, we have seen how to define a contiguous range of loop indices [0, N) with a `RAJA::RangeSegment` object and use it in a RAJA loop execution template to run a loop kernel over the range. For example:

```
RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(0, N), [=] (IdxType i) {
  a[i] += b[i] * c;
});
```

We can accomplish the same result by enumerating the indices in a `RAJA::TypedListSegment` object. Here, we assemble the indices to a standard vector, create a list segment from it, and then pass the list segment to the forall execution template:

```
std::vector<IdxType> idx;
for (IdxType i = 0; i < N; ++i) {
  idx.push_back(i);
}

ListSegType idx_list( &idx[0], idx.size() );

RAJA::forall<RAJA::seq_exec>(idx_list, [=] (IdxType i) {
  a[i] += b[i] * c;
});
```

Note that we are using the following type aliases here:

```
using IdxType = RAJA::Index_type;
using ListSegType = RAJA::TypedListSegment<IdxType>;
```

Recall from discussion in *Indices, Segments, and IndexSets* that `RAJA::Index_type` is a default index type that RAJA defines and which is used in some RAJA constructs as a convenience for users who want a simple mechanism to apply index types consistently.

It is important to note what is really happening when a list segment is used. During loop execution, indices stored in the list segment are passed to the loop body one-by-one, effectively mimicking an indirection array except that the indirection array does not appear in the loop body. For example, we can reverse the order of the indices, run the loop with a new list segment object, and get the same result since the loop is *data-parallel*:

```
std::reverse( idx.begin(), idx.end() );

ListSegType idx_reverse_list( &idx[0], idx.size() );

RAJA::forall<RAJA::seq_exec>(idx_reverse_list, [=] (IdxType i) {
  a[i] += b[i] * c;
});
```

Alternatively, we can also use a RAJA strided range segment to run the loop in reverse by giving it a stride of -1. For example:

```
RAJA::forall<RAJA::seq_exec>(RAJA::RangeStrideSegment(N-1, -1, -1), [=] (IdxType i)
↪{
  a[i] += b[i] * c;
});
```

RAJA IndexSets

The `RAJA::TypedIndexSet` template is a container that can hold any number of segments of arbitrary type. An index set object can be passed to a RAJA loop execution method, just like a segment, to run a loop kernel. When the loop is run, the execution method iterates over the segments and the loop indices in each segment. Thus, the loop iterates can be grouped into different segments to partition the iteration space and iterate over the loop kernel chunks (defined by segments), in serial, in parallel, or in some specific dependency ordering. Individual segments can be executed in serial or parallel.

When an index set is defined, the segment types it may hold must be specified as template arguments. For example, here we create an index set that can hold list segments. Then, we add the first list segment from above to it, and run the loop:

```
RAJA::TypedIndexSet<ListSegType> is1;

is1.push_back( idx_list );  // use list segment created earlier.

RAJA::forall<SEQ_ISET_EXECPOL>(is1, [=] (IdxType i) {
  a[i] += b[i] * c;
});
```

You are probably asking: What is the 'SEQ_ISET_EXECPOL' type used for the execution policy?

Well, it is like execution policy types we have seen up to this point, except that it specifies a two-level policy – one for iterating over the segments and one for executing the iterates defined by each segment. In the example, we specify that we should do each of these operations sequentially by defining the policy as follows:

```
using SEQ_ISET_EXECPOL = RAJA::ExecPolicy<RAJA::seq_segit,
                                          RAJA::seq_exec>;
```

Next, we perform the daxpy operation by partitioning the iteration space into two range segments:

```
RAJA::TypedIndexSet<RAJA::RangeSegment> is2;
is2.push_back( RAJA::RangeSegment(0, N/2) );
is2.push_back( RAJA::RangeSegment(N/2, N) );

RAJA::forall<SEQ_ISET_EXECPOL>(is2, [=] (IdxType i) {
  a[i] += b[i] * c;
});
```

The first range segment is used to run the index range [0, N/2) and the second is used to run the range [N/2, N).

We can also break up the iteration space into three segments, 2 ranges and 1 list:

```
std::vector<IdxType> idx1;
for (IdxType i = N/3; i < 2*N/3; ++i) {
  idx1.push_back(i);
}

ListSegType idx1_list( &idx1[0], idx1.size() );

RAJA::TypedIndexSet<RAJA::RangeSegment, ListSegType> is3;
is3.push_back( RAJA::RangeSegment(0, N/3) );
is3.push_back( idx1_list );
is3.push_back( RAJA::RangeSegment(2*N/3, N) );

RAJA::forall<SEQ_ISET_EXECPOL>(is3, [=] (IdxType i) {
```

(continues on next page)

(continued from previous page)

```
  a[i] += b[i] * c;
});
```

The first range segment runs the index range [0, N/3), the list segment enumerates the indices in the interval [N/3, 2*N/3), and the second range segment runs the range [2*N/3, N). Note that we use the same execution policy as before.

Before we end the discussion of these examples, we demonstrate a few more index set execution policy variations. To run the previous three segment example by iterating over the segments sequentially and executing each segment in parallel using OpenMP multi-threading, we would use this policy definition:

```
using OMP_ISET_EXECPOL1 = RAJA::ExecPolicy<RAJA::seq_segit,
                                           RAJA::omp_parallel_for_exec>;
```

If we wanted to iterate over the segments in parallel using OpenMP multi-threading and execute each segment sequentially, we would use a policy like this:

```
using OMP_ISET_EXECPOL2 = RAJA::ExecPolicy<RAJA::omp_parallel_for_segit,
                                             RAJA::seq_exec>;
```

Finally, to iterate over the segments sequentially and execute each segment in parallel on a GPU by launching a CUDA kernel, we would define this policy:

```
using OMP_ISET_EXECPOL3 = RAJA::ExecPolicy<RAJA::seq_segit,
                                           RAJA::cuda_exec<CUDA_BLOCK_SIZE>>;
```

The file `RAJA/examples/tut_indexset-segments.cpp` contains working code for these examples.

Mesh Vertex Sum Example: Iteration Space Coloring

Key RAJA features shown in this example:

- `RAJA::forall` loop execution template method
- `RAJA::ListSegment` iteration space construct
- `RAJA::IndexSet` iteration space segment container and associated execution policies

The example computes a sum at each vertex on a logically-Cartesian 2D mesh as shown in the figure.

Each sum is an average of the area of the mesh elements that share the vertex. In many "staggered mesh" applications, such an operation is common and is often written in a way that presents the algorithm clearly but prevents parallelization due to potential data races. That is, multiple loop iterates over mesh elements may attempt to write to the same shared vertex memory location at the same time. The example shows how RAJA constructs can be used to enable one to express such an algorithm in parallel and have it run correctly without fundamentally changing how it looks in source code.

After defining the number of elements in the mesh, necessary array offsets and an array that indicates the mapping between an element and its four surrounding vertices, a C-style version of the vertex sum calculation is:

```
for (int j = 0 ; j < N_elem ; ++j) {
  for (int i = 0 ; i < N_elem ; ++i) {
    int ie = i + j*jeoff ;
    int* iv = &(elem2vert_map[4*ie]);
    vertexvol_ref[ iv[0] ] += elemvol[ie] / 4.0 ;
    vertexvol_ref[ iv[1] ] += elemvol[ie] / 4.0 ;
    vertexvol_ref[ iv[2] ] += elemvol[ie] / 4.0 ;
```

(continues on next page)

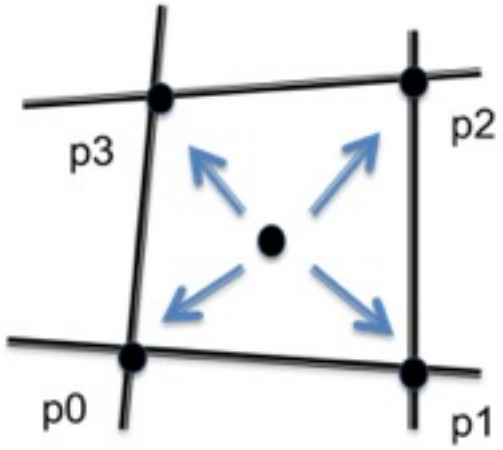

Fig. 5: A portion of the area of each mesh element is summed to the vertices surrounding the element.

(continued from previous page)

```
    vertexvol_ref[ iv[3] ] += elemvol[ie] / 4.0 ;
  }
}
```

RAJA Sequential Variant

A nested loop RAJA variant of this kernel is:

```
using EXEC_POL1 =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::seq_exec,     // j
      RAJA::statement::For<0, RAJA::seq_exec,   // i
        RAJA::statement::Lambda<0>
      >
    >
  >;

RAJA::kernel<EXEC_POL1>( RAJA::make_tuple(RAJA::RangeSegment(0, N_elem),
                                          RAJA::RangeSegment(0, N_elem)),
  [=](int i, int j) {
    int ie = i + j*jeoff ;
    int* iv = &(elem2vert_map[4*ie]);
    vertexvol[ iv[0] ] += elemvol[ie] / 4.0 ;
    vertexvol[ iv[1] ] += elemvol[ie] / 4.0 ;
    vertexvol[ iv[2] ] += elemvol[ie] / 4.0 ;
    vertexvol[ iv[3] ] += elemvol[ie] / 4.0 ;
});
```

Note that this version cannot be guaranteed to run correctly in parallel by simply changing the loop execution policies as we have done in other examples. We would like to use RAJA to enable parallel execution and without changing the way the kernel looks in source code. By applying a RAJA index set and suitably-defined list segments, we can accomplish this.

RAJA Parallel Variants

To enable the kernel to run safely in parallel, by eliminating the race conditions, we partition the element iteration space into four subsets (or *colors*) indicated by the numbers in the figure below, which represents a portion of our logically-Cartesian 2D mesh.

2	3	2	3
0	1	0	1
2	3	2	3
0	1	0	1

Note that none of the elements with the same number share a common vertex. Thus, we can iterate over all elements with the same number (i.e., color) in parallel.

First, we define four vectors to gather the mesh element indices for each color:

```
std::vector<int> idx0;
std::vector<int> idx1;
std::vector<int> idx2;
std::vector<int> idx3;

for (int j = 0 ; j < N_elem ; ++j) {
  for (int i = 0 ; i < N_elem ; ++i) {
    int ie = i + j*jeoff ;
    if ( i % 2 == 0 ) {
      if ( j % 2 == 0 ) {
        idx0.push_back(ie);
      } else {
        idx2.push_back(ie);
      }
    } else {
      if ( j % 2 == 0 ) {
        idx1.push_back(ie);
      } else {
        idx3.push_back(ie);
      }
    }
  }
}
```

Then, we create a RAJA index set with four list segments, one for each color, using the vectors:

```
using SegmentType = RAJA::TypedListSegment<int>;

RAJA::TypedIndexSet<SegmentType> colorset;

colorset.push_back( SegmentType(&idx0[0], idx0.size()) );
colorset.push_back( SegmentType(&idx1[0], idx1.size()) );
colorset.push_back( SegmentType(&idx2[0], idx2.size()) );
colorset.push_back( SegmentType(&idx3[0], idx3.size()) );
```

Now, we can use an index set execution policy that iterates over the segments sequentially and executes each segment in parallel using OpenMP multi-threading (and `RAJA::forall`):

```
using EXEC_POL3 = RAJA::ExecPolicy<RAJA::seq_segit,
                                   RAJA::omp_parallel_for_exec>;
```

(continues on next page)

(continued from previous page)

```
RAJA::forall<EXEC_POL3>(colorset, [=](int ie) {
  int* iv = &(elem2vert_map[4*ie]);
  vertexvol[ iv[0] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[1] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[2] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[3] ] += elemvol[ie] / 4.0 ;
});
```

We no longer need to use the offset variable to compute the element index in terms of 'i' and 'j' since the loop is no longer nested and the element indices are directly encoded in the list segments.

For completeness, here is the RAJA variant where we iterate over the segments sequentially, and execute each segment in parallel via a CUDA kernel launch on a GPU:

```
using EXEC_POL4 = RAJA::ExecPolicy<RAJA::seq_segit,
                                   RAJA::cuda_exec<CUDA_BLOCK_SIZE>>;

RAJA::forall<EXEC_POL4>(colorset, [=] RAJA_DEVICE (int ie) {
  int* iv = &(elem2vert_map[4*ie]);
  vertexvol[ iv[0] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[1] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[2] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[3] ] += elemvol[ie] / 4.0 ;
});
```

Here, we have marked the lambda loop body with the 'RAJA_DEVICE' macro and specified the number of threads in a CUDA thread block in the segment execution policy.

The file `RAJA/examples/tut_vertexsum-coloring.cpp` contains the complete working example code.

Reductions

Key RAJA features shown in this example:

- `RAJA::forall` loop execution template
- `RAJA::RangeSegment` iteration space construct
- RAJA reduction types
- RAJA reduction policies

In the *Vector Dot Product (Sum Reduction)* example, we showed how to use the RAJA sum reduction type. The following example uses all supported RAJA reduction types: min, max, sum, min-loc, max-loc.

Note: Multiple RAJA reductions can be combined in any RAJA loop kernel execution method, and reduction operations can be combined with any other kernel operations.

We start by allocating an array (the memory manager in the example uses CUDA Unified Memory if CUDA is enabled) and initializing its values in a manner that makes the example mildly interesting and able to show what the different reduction types do. Specifically, the array is initialized to a sequence of alternating values ('1' and '-1'). Then, two values near the middle of the array are set to '-100' and '100':

```
//
// Define array length
//
  const int N = 1000000;

//
// Allocate array data and initialize data to alternating sequence of 1, -1.
//
  int* a = memoryManager::allocate<int>(N);

  for (int i = 0; i < N; ++i) {
    if ( i % 2 == 0 ) {
      a[i] = 1;
    } else {
      a[i] = -1;
    }
  }

//
// Set min and max loc values
//
  const int minloc_ref = N / 2;
  a[minloc_ref] = -100;

  const int maxloc_ref = N / 2 + 1;
  a[maxloc_ref] = 100;
```

We also define a range segment to iterate over the array:

```
  RAJA::RangeSegment arange(0, N);
```

With these parameters and data initialization, all the code examples presented below will generate the following results:

- the sum will be zero
- the min will be -100
- the max will be 100
- the min loc will be N/2
- the max loc will be N/2 + 1

A sequential kernel that exercises all RAJA sequential reduction types is:

```
  using EXEC_POL1   = RAJA::seq_exec;
  using REDUCE_POL1 = RAJA::seq_reduce;

  RAJA::ReduceSum<REDUCE_POL1, int> seq_sum(0);
  RAJA::ReduceMin<REDUCE_POL1, int> seq_min(std::numeric_limits<int>::max());
  RAJA::ReduceMax<REDUCE_POL1, int> seq_max(std::numeric_limits<int>::min());
  RAJA::ReduceMinLoc<REDUCE_POL1, int> seq_minloc(std::numeric_limits<int>::max(), -
↪1);
  RAJA::ReduceMaxLoc<REDUCE_POL1, int> seq_maxloc(std::numeric_limits<int>::min(), -
↪1);

  RAJA::forall<EXEC_POL1>(arange, [=](int i) {

    seq_sum += a[i];
```

(continues on next page)

(continued from previous page)

```
  seq_min.min(a[i]);
  seq_max.max(a[i]);

  seq_minloc.minloc(a[i], i);
  seq_maxloc.maxloc(a[i], i);

});

std::cout << "\tsum = " << seq_sum.get() << std::endl;
std::cout << "\tmin = " << seq_min.get() << std::endl;
std::cout << "\tmax = " << seq_max.get() << std::endl;
std::cout << "\tmin, loc = " << seq_minloc.get() << " , "
                             << seq_minloc.getLoc() << std::endl;
std::cout << "\tmax, loc = " << seq_maxloc.get() << " , "
                             << seq_maxloc.getLoc() << std::endl;
```

Note that each reduction object takes an initial value at construction. Also, within the kernel, updating each reduction is done via an operator or method that is basically what you would expect (i.e., '+=' for sum, 'min()' for min, etc.). After the kernel executes, the reduced value computed by each reduction object is retrieved after the kernel by calling a 'get()' method on the reduction object. The min-loc/max-loc index values are obtained using 'getLoc()' methods.

For parallel multi-threading execution via OpenMP, the example can be run by replacing the execution and reduction policies with:

```
using EXEC_POL2   = RAJA::omp_parallel_for_exec;
using REDUCE_POL2 = RAJA::omp_reduce;
```

Similarly, the kernel containing the reductions can be run in parallel on a CUDA GPU using these policies:

```
using EXEC_POL3   = RAJA::cuda_exec<CUDA_BLOCK_SIZE>;
using REDUCE_POL3 = RAJA::cuda_reduce;
```

Note: Each RAJA reduction type requires a reduction policy that must be compatible with the execution policy for the kernel in which it is used.

The file `RAJA/examples/tut_reductions.cpp` contains the complete working example code.

Computing a Histogram with Atomic Operations

Key RAJA features shown in this example:

- `RAJA::forall` loop execution template
- `RAJA::RangeSegment` iteration space construct
- RAJA atomic add operation

The example uses an integer array of length 'N' randomly initialized with values in the interval [0, M). While iterating over the array, the kernel accumulates the number of occurrences of each value in the array using atomic add operations. Atomic operations allow one to update a memory location referenced by a specific address in parallel without data races. The example shows how to use RAJA portable atomic operations and that they are used similarly for different programming model back-ends.

Note: Each RAJA reduction operation requires an atomic policy type parameter that must be compatible with the execution policy for the kernel in which it is used.

For a complete description of supported RAJA atomic operations and atomic policies, please see *Atomics*.

All code snippets described below use the loop range:

```
RAJA::TypedRangeSegment<int> array_range(0, N);
```

and the integer array 'bins' of length 'M' to accumulate the number of occurrences of each value in the array.

Here is the OpenMP version:

```
using EXEC_POL2 = RAJA::omp_parallel_for_exec;
using ATOMIC_POL2 = RAJA::atomic::omp_atomic;

RAJA::forall<EXEC_POL2>(array_range, [=](int i) {

  RAJA::atomic::atomicAdd<ATOMIC_POL2>(&bins[array[i]], 1);

});
```

Each slot in the 'bins' array is incremented by one when a value associated with that slot is encountered. Note that the `RAJA::atomic::atomicAdd` operation uses an OpenMP atomic policy, which is compatible with the OpenMP loop execution policy.

The CUDA version is similar:

```
using EXEC_POL4 = RAJA::cuda_exec<CUDA_BLOCK_SIZE>;
using ATOMIC_POL4 = RAJA::atomic::cuda_atomic;

RAJA::forall<EXEC_POL4>(array_range, [=] RAJA_DEVICE(int i) {

  RAJA::atomic::atomicAdd<ATOMIC_POL4>(&bins[array[i]], 1);

});
```

Here, the atomic add operation uses a CUDA atomic policy, which is compatible with the CUDA loop execution policy.

Note that RAJA provides an `auto_atomic` policy for easier usage and improved portability. This policy will do the right thing in most circumstances. Specifically, if it is encountered in a CUDA execution context, the CUDA atomic policy will be applied. If OpenMP is enabled, the OpenMP atomic policy will be used, which should be correct in a sequential execution context as well. Otherwise, the sequential atomic policy will be applied.

For example, here is the CUDA version that uses the 'auto' atomic policy:

```
using ATOMIC_POL5 = RAJA::atomic::auto_atomic;

RAJA::forall<EXEC_POL4>(array_range, [=] RAJA_DEVICE(int i) {

  RAJA::atomic::atomicAdd<ATOMIC_POL5>(&bins[array[i]], 1);

});
```

The same CUDA loop execution policy as in the previous example is used.

The file `RAJA/examples/tut_atomic-binning.cpp` contains the complete working example code.

Parallel Scan Operations

Key RAJA features shown in this section:

- `RAJA::inclusive_scan` operation
- `RAJA::inclusive_scan_inplace` operation
- `RAJA::exclusive_scan` operation
- `RAJA::exclusive_scan_inplace` operation
- RAJA operators for different types of scans; e.g., plus, minimum, maximum, etc.

Below, we present examples of RAJA sequential, OpenMP, and CUDA scan operations and show how different scan operations can be performed by passing different RAJA operators to the RAJA scan template methods. Each operator is a template type, where the template argument is the type of the values it operates on. For a summary of RAJA scan functionality, please see *Parallel Scan Operations*.

Note: RAJA scan operations use the same execution policy types that `RAJA::forall` loop execution templates do.

Each of the examples below uses the same integer arrays for input and output values. We set the input array and print them as follows:

```
//
// Define array length
//
  const int N = 20;

//
// Allocate and array vector data
//
  int* in = memoryManager::allocate<int>(N);
  int* out = memoryManager::allocate<int>(N);

  std::iota(in, in+N, -1);

  std::shuffle(in, in + N, std::mt19937{std::random_device{}()});
  std::cout << "\n in values...\n";
  printArray(in, N);
  std::cout << "\n";
```

This generates the following sequence of values in the 'in' array:

```
3 -1 2 15 7 5 17 9 6 18 1 10 0 14 13 4 11 12 8 16
```

Inclusive Scans

A sequential inclusive scan operation is performed by:

```
  RAJA::inclusive_scan<RAJA::seq_exec>(in, in + N, out);
```

Since no operator is passed to the scan method, the default 'sum' operation is applied and the result generated in the 'out' array is a prefix-sum based on the 'in' array. The resulting 'out' array contains the values:

```
3 2 4 19 26 31 48 57 63 81 82 92 92 106 119 123 134 146 154 170
```

We can be explicit about the operation used in the scan by passing the 'plus' operator to the scan method:

```
RAJA::inclusive_scan<RAJA::seq_exec>(in, in + N, out,
                                     RAJA::operators::plus<int>{});
```

The result in the 'out' array is the same.

An inclusive parallel scan operation using OpenMP multi-threading is accomplished similarly by replacing the execution policy type:

```
RAJA::inclusive_scan<RAJA::omp_parallel_for_exec>(in, in + N, out,
                                                  RAJA::operators::plus<int>{});
```

As is commonly done with RAJA, the only difference between this code and the previous one is that the execution policy is different. If we want to run the scan on a GPU using CUDA, we would use a CUDA execution policy. This will be shown shortly.

Exclusive Scans

A sequential exclusive scan (plus) operation is performed by:

```
RAJA::exclusive_scan<RAJA::seq_exec>(in, in + N, out,
                                     RAJA::operators::plus<int>{});
```

This generates the following sequence of values in the output array:

```
0 3 2 4 19 26 31 48 57 63 81 82 92 92 106 119 123 134 146 154
```

Note that the exclusive scan result is different than the inclusive scan result in two ways. The first entry in the result is the *identity* of the operator used (here, it is zero, since the operator is 'plus') and, after that, the output sequence is shifted one position to the right.

Running the same scan operation on a GPU using CUDA is done by:

```
RAJA::exclusive_scan<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(in, in + N, out,
                                     RAJA::operators::plus<int>{});
```

Note that we pass the number of threads per CUDA thread block as the template argument to the CUDA execution policy as we do in other cases.

In-place Scans and Other Operators

In-place scan operations generate the same results as the scan operations we have just described. However, the result is generated in the input array directly so **only one array is passed to in-place scan methods.**

Here is a sequential inclusive in-place scan that uses the 'minimum' operator:

```
std::copy_n(in, N, out);

RAJA::inclusive_scan_inplace<RAJA::seq_exec>(out, out + N,
                                             RAJA::operators::minimum<int>{});
```

Note that, before the scan, we copy the input array into the output array so the result is generated in the output array. Doing this, we avoid having to re-initialize the input array to use it in other examples.

This generates the following sequence in the output array:

```
3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Here is a sequential exclusive in-place scan that uses the 'maximum' operator:

```
  std::copy_n(in, N, out);

  RAJA::exclusive_scan_inplace<RAJA::seq_exec>(out, out + N,
                                               RAJA::operators::maximum<int>{});
```

This generates the following sequence in the output array:

```
-2147483648 3 3 3 15 15 15 17 17 17 18 18 18 18 18 18 18 18 18 18
```

Note that the first value in the result is the negative of the max int value; i.e., the identity of the maximum operator.

As you may expect at this point, running an exclusive in-place prefix-sum operation using OpenMP is accomplished by:

```
  std::copy_n(in, N, out);

  RAJA::exclusive_scan_inplace<RAJA::omp_parallel_for_exec>(out, out + N,
                                                            RAJA::operators::plus<int>
↪{});
```

This generates the following sequence in the output array (as we saw earlier):

```
0 3 2 4 19 26 31 48 57 63 81 82 92 92 106 119 123 134 146 15
```

Lastly, we show a parallel inclusive in-place prefix-sum operation using CUDA:

```
  std::copy_n(in, N, out);

  RAJA::inclusive_scan_inplace<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(out, out + N,
                                               RAJA::operators::plus<int>{});
```

The file `RAJA/examples/tut_scan.cpp` contains the complete working example code.

3.4.4 Complex Loops: Transformations and Advanced RAJA Features

The examples in this section illustrate how to use `RAJA::kernel` methods to execute complex loop kernels, such as nested loops. It also describes how to construct kernel execution policies, use different view types and tiling mechanisms to transform loop patterns.

Matrix Multiplication (Nested Loops)

Key RAJA features shown in the following examples:

- `RAJA::kernel` template for nested-loop execution
- RAJA kernel execution policies
- `RAJA::View` multi-dimensional data access
- Basic RAJA nested-loop interchange

In this example, we present different ways to perform multiplication of two square matrices 'A' and 'B' of dimension N x N and store the result in matrix 'C'. To motivate the use of the `RAJA::View` abstraction that we use, we define the following macros to access the matrix entries in the C-version:

```
#define A(r, c) A[c + N * r]
#define B(r, c) B[c + N * r]
#define C(r, c) C[c + N * r]
```

Then, a typical C-style sequential matrix multiplication operation looks like the following:

```
for (int row = 0; row < N; ++row) {
  for (int col = 0; col < N; ++col) {

    double dot = 0.0;
    for (int k = 0; k < N; ++k) {
      dot += A(row, k) * B(k, col);
    }

    C(row, col) = dot;
  }
}
```

For the RAJA variants of the matrix multiple operation, we use `RAJA::Range Segment` objects to define the matrix row and column and dot product iteration spaces:

```
RAJA::RangeSegment row_range(0, N);
RAJA::RangeSegment col_range(0, N);
RAJA::RangeSegment dot_range(0, N);
```

We also use `RAJA::View` objects, which allow us to access matrix entries in a multi-dimensional manner similar to the C-style version that uses macros. We create a two-dimensional N x N 'view' for each of the three matrices:

```
RAJA::View<double, RAJA::Layout<DIM>> Aview(A, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Bview(B, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Cview(C, N, N);
```

Although we only show very basic RAJA view usage here, RAJA views can be used to encapsulate a variety of different data layouts and access patterns, including permutations, strides, etc. For more information about RAJA views, see *View and Layout*.

Should I Use RAJA::forall For Nested Loops?

We begin by walking through some RAJA versions of the matrix multiplication operation that show RAJA usage that **we do not recommend**, but which helps to motivate the `RAJA::kernel` interface. We noted some rationale behind this preference in *Complex Loops (RAJA::kernel)*. Here, we discuss this in more detail.

Starting with the C-style kernel above, we first convert the outermost 'row' loop to a `RAJA::forall` method call with a sequential execution policy:

```
RAJA::forall<RAJA::loop_exec>( row_range, [=](int row) {

  for (int col = 0; col < N; ++col) {

    double dot = 0.0;
    for (int k = 0; k < N; ++k) {
      dot += Aview(row, k) * Bview(k, col);
```

(continues on next page)

(continued from previous page)

```
    }

    Cview(row, col) = dot;

  }

});
```

Here, the lambda expression for the loop body contains the inner 'col' and 'k' loops.

Note that changing the RAJA execution policy to an OpenMP or CUDA policy enables the outer 'row' loop to run in parallel. When this is done, each thread executes the lambda expression body, which contains the 'col' and 'k' loops. Although this enables some parallelism, there is still more available. In a bit, we will how the `RAJA::kernel` interface helps us to expose all available parallelism.

Next, we nest a `RAJA::forall` method call for the 'column' loop inside the outer lambda expression:

```
RAJA::forall<RAJA::loop_exec>( row_range, [=](int row) {

  RAJA::forall<RAJA::loop_exec>( col_range, [=](int col) {

    double dot = 0.0;
    for (int k = 0; k < N; ++k) {
      dot += Aview(row, k) * Bview(k, col);
    }

    Cview(row, col) = dot;

  });

});
```

Here, the innermost lambda expression contains the row-column dot product initialization, the inner 'k' loop for the dot product, and the operation that assigns the dot product to the proper location in the result matrix.

Note that we can replace either RAJA execution policy with an OpenMP execution policy to parallelize either the 'row' or 'col' loop. For example, we can use an OpenMP execution policy on the outer 'row' loop and the result will be the same as using an OpenMP policy in the `RAJA::forall` statement in the previous case.

We do not recommend using a parallel execution policy for both loops in this type of kernel as the results may not be what is expected and RAJA provides better mechanisms for parallelizing nested loops. Also, changing the outer loop policy to a CUDA policy will not compile. This is by design in RAJA since nesting forall statements inside lambdas in this way has limited utility, is inflexible, and can hinder performance when compared to `RAJA::kernel` constructs, which we describe next.

Basic RAJA::kernel Variants

Next, we show how to cast the matrix-multiplication operation using `RAJA::kernel` nested-loop capabilities, which were introduced in *Complex Loops (RAJA::kernel)*. We first present a complete example, and then describe its key elements, noting important differences between `RAJA::kernel` and `RAJA::forall` loop execution interfaces.

```
using EXEC_POL =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::loop_exec,     // row
      RAJA::statement::For<0, RAJA::loop_exec,  // col
```

(continues on next page)

(continued from previous page)

```
        RAJA::statement::Lambda<0>
      >
    >
  >;

RAJA::kernel<EXEC_POL>(RAJA::make_tuple(col_range, row_range),
  [=](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;

});
```

Here, we use `RAJA::kernel` to express the outer 'row' and 'col' loops; the inner 'k' loop is included in the lambda expression for the loop body. Note that the `RAJA::kernel` template takes two arguments. Similar to `RAJA::forall`, the first argument describes the iteration space and the second argument is the lambda loop body. Unlike `RAJA::forall`, the iteration space for `RAJA::kernel` is defined as a *tuple* of ranges (created via the `RAJA::make_tuple` method), one for the 'col' loop and one for the 'row' loop. Also, the lambda expression takes an iteration index argument for entry in the iteration space tuple.

Note: The number and order of lambda arguments must match the number and order of the elements in the tuple for this to be correct.

Another important difference between `RAJA::forall` and `RAJA::kernel` is in the execution policy template parameter. The execution policy defined by the `RAJA::KernelPolicy` type used here specifies a policy for each level in the loop nest via nested `RAJA::statement::For` types. Here, the row and column loops will both execute sequentially. The integer that appears as the first template parameter to each 'For' statement corresponds to the position of a range in the iteration space tuple and also to the associated iteration index argument to the lambda. Here, '0' is the 'col' range and '1' is the 'row' range because that is the order those ranges appear in the tuple. The innermost type `RAJA::statement::Lambda<0>` indicates that the first lambda expression (the only one in this case!) argument passed to the `RAJA::kernel` method after the index space tuple will be invoked inside the nested loops.

The integer arguments to the `RAJA::statement::For` types are needed to enable a variety of kernel execution patterns and transformations. Since the kernel policy is a single unified construct, it can be used to parallelize the nested loop iterations together, which we will show later. Also, the levels in the loop nest can be permuted by reordering the policy arguments; this is analogous to how one would reorder C-style nested loops; i.e., reorder for-statements for each loop nest level. These execution patterns and transformations can be achieved by changing the policy and leaving the loop kernel code as is.

If we want to execute the row loop using OpenMP multi-threaded parallelism and keep the column loop sequential, the policy we would use is:

```
using EXEC_POL1 =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::omp_parallel_for_exec,  // row
      RAJA::statement::For<0, RAJA::loop_exec,            // col
        RAJA::statement::Lambda<0>
      >
    >
  >;
```

To swap the loop nest ordering and keep the same execution policy on each loop, we would use the following policy, which swaps the `RAJA::statement::For` types. The inner loop is now the 'row' loop and is run in parallel; the outer loop is now the 'col' loop and is still sequential:

```
using EXEC_POL2 =
  RAJA::KernelPolicy<
    RAJA::statement::For<0, RAJA::loop_exec,                  // col
      RAJA::statement::For<1, RAJA::omp_parallel_for_exec,    // row
        RAJA::statement::Lambda<0>
      >
    >
  >;
```

Note: It is important to note that these kernel transformations, and others, can be done by switching the `RAJA::KernelPolicy` type with no changes to the loop kernel code.

In *Nested Loop Interchange*, we provide a more detailed discussion of the mechanics of loop nest reordering. Next, we show other variations of the matrix multiplication kernel that illustrate other `RAJA::kernel` features.

More Complex RAJA::kernel Variants

The matrix multiplication kernel variations described in this section use execution policies to express the outer row and col loops as well as the inner dot product loop using the RAJA kernel interface. They illustrate more complex policy examples and show additional RAJA kernel features.

The first example uses sequential execution for all loops:

```
using EXEC_POL6 =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::loop_exec,
      RAJA::statement::For<0, RAJA::loop_exec,
        RAJA::statement::Lambda<0>,  // dot = 0.0
        RAJA::statement::For<2, RAJA::loop_exec,
          RAJA::statement::Lambda<1> // inner loop: dot += ...
        >,
        RAJA::statement::Lambda<2>   // set C(row, col) = dot
      >
    >
  >;

RAJA::kernel_param<EXEC_POL6>(
  RAJA::make_tuple(col_range, row_range, dot_range),

  RAJA::tuple<double>{0.0},    // thread local variable for 'dot'

  // lambda 0
  [=] (int /* col */, int /* row */, int /* k */, double& dot) {
     dot = 0.0;
  },

  // lambda 1
  [=] (int col, int row, int k, double& dot) {
     dot += Aview(row, k) * Bview(k, col);
  },
```

(continues on next page)

(continued from previous page)

```
    // lambda 2
    [=] (int col, int row, int /* k */, double& dot) {
       Cview(row, col) = dot;
    }

  );
```

Note that we use a `RAJA::kernel_param` method to execute the kernel. It is similar to `RAJA::kernel` except that it accepts a tuple as its second argument (between the iteration space tuple and the lambda expressions). The tuple is a set of *parameters* that can be used in the kernel to pass data into lambda expressions. Here, the parameter tuple holds a single scalar thread-local variable for the dot product.

The remaining arguments include a sequence of lambda expressions representing different parts of the inner loop body. We use three lambda expressions that: initialize the dot product variable (lambda 0), define the 'k' inner loop row-col dot product operations (lambda 1), and store the computed row-col dot product in the proper location in the result matrix (lambda 2). Note that all lambdas take the same arguments in the same order, which is required for the kernel to be well-formed. In addition to the loop index variables, we pass the scalar dot product variable into each lambda. This enables the same variable to be used in all three lambdas. However, also note that not all lambda expressions use all three index variables. They are declared, but left unnamed to prevent compiler warnings.

The execution policy type passed to the `RAJA::kernel_param` method as a template parameter describes how the statements and lambda expressions are assembled to form the complete kernel. To illustrate this, we describe various policies that enable the kernel to run in different ways. In each case, the `RAJA::kernel_param` method call, including its arguments is the same. The curious reader will inspect the example code to see that this is indeed the case.

Next, we show how to collapse nested loops in an OpenMP parallel region using a `RAJA::statement::Collapse` type in the execution policy. This allows one to parallelize multiple levels in a loop nest using OpenMP directives, for instance. The following policy will collapse the two outer loops:

```
using EXEC_POL7 =
  RAJA::KernelPolicy<
    RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec,
                              RAJA::ArgList<1, 0>,   // row, col
      RAJA::statement::Lambda<0>,  // dot = 0.0
      RAJA::statement::For<2, RAJA::loop_exec,
        RAJA::statement::Lambda<1> // inner loop: dot += ...
      >,
      RAJA::statement::Lambda<2>   // set C(row, col) = dot
    >
  >;
```

The `RAJA::ArgList` type indicates which loops in the nest are to be collapsed and their nesting order within the collapse region. The integers passed to `ArgList` are indices of entries in the tuple of iteration spaces and indicate inner to outer loop levels when read from right to left (i.e., here '1, 0' indicates the column loop is the inner loop and the row loop is the outer). For this transformation there are no `statement::For` types and policies for the individual loop levels inside the OpenMP collapse region.

Lastly, we describe how to use `RAJA::statement::CudaKernel` types to generate a CUDA kernel launched with a particular thread-block decomposition. We reiterate that although the policies are different, the kernels themselves are identical to the sequential and OpenMP variants above.

Here is a policy that will distribute the row indices across CUDA thread blocks and column indices across threads in each block:

```
  using EXEC_POL8 =
    RAJA::KernelPolicy<
      RAJA::statement::CudaKernel<
        RAJA::statement::For<1, RAJA::cuda_block_y_loop,     // row
          RAJA::statement::For<0, RAJA::cuda_thread_x_loop, // col
            RAJA::statement::Lambda<0>,   // dot = 0.0
            RAJA::statement::For<2, RAJA::seq_exec,
                RAJA::statement::Lambda<1> // dot += ...
            >,
            RAJA::statement::Lambda<2>   // set C = ...
          >
        >
      >
    >;
```

This is equivalent to defining a CUDA kernel with the lambda body inside it and defining row and column indices as:

```
int row = blockIdx.x;
int col = threadIdx.x;
```

and launching the kernel with appropriate CUDA grid and thread-block dimensions.

The following policy will tile row and col indices across two-dimensional CUDA thread blocks with 'x' and 'y' dimensions defined by a 'CUDA_BLOCK_SIZE' parameter that can be set at compile time. Within each tile, the kernel iterates are executed by CUDA threads.

```
  using EXEC_POL9 =
    RAJA::KernelPolicy<
      RAJA::statement::CudaKernel<
        RAJA::statement::Tile<1, RAJA::statement::tile_fixed<CUDA_BLOCK_SIZE>,␣
↪RAJA::cuda_block_y_loop,
          RAJA::statement::Tile<0, RAJA::statement::tile_fixed<CUDA_BLOCK_SIZE>,␣
↪RAJA::cuda_block_x_loop,
            RAJA::statement::For<1, RAJA::cuda_thread_y_loop, // row
              RAJA::statement::For<0, RAJA::cuda_thread_x_loop, // col
                RAJA::statement::Lambda<0>,   // dot = 0.0
                RAJA::statement::For<2, RAJA::seq_exec,
                    RAJA::statement::Lambda<1> // dot += ...
                >,
                RAJA::statement::Lambda<2>   // set C = ...
              >
            >
          >
        >
      >
    >;
```

Note that the tiling mechanism requires a `RAJA::statement::Tile` type, with a tile size and a tiling execution policy, plus a `RAJA::statement::For` type with an execution execution policy for each tile dimension.

In *Tiled Matrix Transpose* and *Matrix Transpose with Local Array*, we will discuss loop tiling in more detail including how it can be used to improve performance of certain algorithms.

The file `RAJA/examples/tut_matrix-multiply.cpp` contains the complete working code for all examples described in this section. It also contains a raw CUDA version of the kernel for comparison.

Nested Loop Interchange

Key RAJA features shown in this example:

- `RAJA::kernel` loop iteration templates
- RAJA nested loop execution policies
- Nested loop reordering (i.e., loop interchange)
- RAJA strongly-types indices

In *Complex Loops (RAJA::kernel)*, we introduced the basic mechanics in RAJA for representing nested loops. In *Matrix Multiplication (Nested Loops)*, we presented a complete example using RAJA nested loop features. The following example shows the nested loop interchange process in more detail. Specifically, we describe how to reorder nested policy arguments and introduce strongly-typed index variables that can help users write correct nested loop code with RAJA. The example does not perform any actual computation; each kernel simply prints out the loop indices in the order that the iteration spaces are traversed. Thus, only sequential execution policies are used. However, the mechanics work the same way for other RAJA execution policies.

Before we dive into the example, we note important features applied here that represent the main differences between nested-loop RAJA and the `RAJA::forall` loop construct for simple (i.e., non-nested) loops:

- An index space (e.g., range segment) and lambda index argument are required for each level in a loop nest. This example contains triply-nested loops, so there will be three ranges and three index arguments.
- The index spaces for the nested loop levels are specified in a RAJA tuple object. The order of spaces in the tuple must match the order of index arguments to the lambda for this to be correct, in general. RAJA provides strongly-typed indices to help with this, which we show here.
- An execution policy is required for each level in a loop nest. These are specified as nested statements in the `RAJA::KernelPolicy` type.
- The loop nest ordering is specified in the nested kernel policy – the first `statement::For` type identifies the outermost loop, the second `statement::For` type identifies the loop nested inside the outermost loop, and so on.

We begin by defining three named **strongly-typed** variables for the loop index variables.

```
RAJA_INDEX_VALUE(KIDX, "KIDX");
RAJA_INDEX_VALUE(JIDX, "JIDX");
RAJA_INDEX_VALUE(IIDX, "IIDX");
```

We also define three **typed** range segments which bind the ranges to the index variable types via template specialization:

```
  RAJA::TypedRangeSegment<KIDX> KRange(2, 4);
  RAJA::TypedRangeSegment<JIDX> JRange(1, 3);
  RAJA::TypedRangeSegment<IIDX> IRange(0, 2);
```

When these features are used as in this example, the compiler will generate error messages if the lambda expression index argument ordering and types do not match the index ordering in the tuple.

We present a complete example, and then describe its key elements:

```
  using KJI_EXECPOL = RAJA::KernelPolicy<
                        RAJA::statement::For<2, RAJA::seq_exec,     // k
                          RAJA::statement::For<1, RAJA::seq_exec,  // j
                            RAJA::statement::For<0, RAJA::seq_exec,// i
                              RAJA::statement::Lambda<0>
                            >
```

(continues on next page)

(continued from previous page)

```
                          >
                        >
                      >;

  RAJA::kernel<KJI_EXECPOL>( RAJA::make_tuple(IRange, JRange, KRange),
  [=] (IIDX i, JIDX j, KIDX k) {
     printf( " (%d, %d, %d) \n", (int)(*i), (int)(*j), (int)(*k));
  });
```

Here, the `RAJA::kernel` execution template takes two arguments: a tuple of ranges, one for each level in the loop nest, and the lambda expression loop body. Note that the lambda has an index argument for each range and that their order and types match.

The execution policy for the loop nest is specified in the `RAJA::KernelPolicy` type. Each level in the loop nest is identified by a `statement::For` type, which identifies the iteration space and execution policy for the level. Here, each level uses a sequential execution policy. This is for illustration purposes; if you run the example code, you will see the loop index triple printed in the exact order in which the loops are runs. The integer that appears as the first template argument to each `statement::For` type corresponds to the index of a range in the tuple and also to the associated lambda index argument; i.e., '0' is for 'i', '1' is for 'j', and '2' is for 'k'. The integer arguments are needed so that the levels in the loop nest can be reordered by changing the policy while the kernel remains the same.

Here, the 'k' index corresponds to the outermost loop (slowest index), the 'j' index corresponds to the middle loop, and the 'i' index is for the innermost loop (fastest index). In other words, if written using C-style for-loops, the loop would appear as:

```
for (int k = 2; k< 4; ++k) {
  for (int j = 1; j < 3; ++j) {
    for (int i = 0; j < 2; ++i) {
      // print loop index triple...
    }
  }
}
```

Next we permute the loop nest ordering so that the 'j' loop is the outermost, the 'i' loop is in the middle, and the 'k' loop is the innermost with the following policy:

```
  using JIK_EXECPOL = RAJA::KernelPolicy<
                        RAJA::statement::For<1, RAJA::seq_exec,     // j
                          RAJA::statement::For<0, RAJA::seq_exec,  // i
                            RAJA::statement::For<2, RAJA::seq_exec,// k
                              RAJA::statement::Lambda<0>
                            >
                          >
                        >
                      >;
```

Note that we have simply reordered the nesting of the `RAJA::statement::For` types. This is analogous to reordering 'for' statements in traditional C-style nested loops. Here, the analogous C-style loop nest would appear as:

```
for (int j = 1; j < 3; ++j) {
  for (int i = 0; i < 2; ++i) {
    for (int k = 2; k< 4; ++k) {
      // print loop index triple...
    }
  }
}
```

Finally, we permute the loops again so that the 'i' loop is the outermost, the 'k' loop is in the middle, and the 'j' loop is the innermost with the following policy:

```
  using IKJ_EXECPOL = RAJA::KernelPolicy<
                        RAJA::statement::For<0, RAJA::seq_exec,     // i
                          RAJA::statement::For<2, RAJA::seq_exec,  // k
                            RAJA::statement::For<1, RAJA::seq_exec,// j
                              RAJA::statement::Lambda<0>
                            >
                          >
                        >
                      >;
```

For completeness, analogous C-style loop nest would appear as:

```
for (int i = 0; j < 2; ++i) {
  for (int k = 2; k< 4; ++k) {
    for (int j = 1; j < 3; ++j) {
      // print loop index triple...
    }
  }
}
```

Hopefully, it should be clear how this works at this point. If not, the typed indices and typed range segments can help by enabling the compiler to let you know when something is not correct.

For example, this version of the loop will generate a compilation error (note that the kernel execution policy is the same as in the previous example):

```
  RAJA::kernel<IKJ_EXECPOL>( RAJA::make_tuple(IRange, JRange, KRange),
  [=] (JIDX i, IIDX j, KIDX k) {
     printf( " (%d, %d, %d) \n", (int)(*i), (int)(*j), (int)(*k));
  });
```

If you carefully compare the range ordering in the tuple to the lambda argument types, you will see what's wrong.

Do you see the problem?

The file `RAJA/examples/tut_nested-loop-reorder.CPU` contains the complete working example code.

Batched Matrix-Multiply (Permuted Layouts)

Key RAJA features shown in the following example:

- `RAJA::forall` loop traversal template
- RAJA execution policies
- `RAJA::View` multi-dimensional data access
- `RAJA::make_permuted_layout` method to permute data ordering

This example performs batched matrix multiplication for a set of 3×3 matrices using two different data layouts.

Matrices A and B are multiplied with the product stored in matrix C. The notation A^e_{rc} indicates the row r and column c entry of matrix e. We describe the two data layouts we use for two matrices. The extension to more than two matrices is straightforward. Using different data layouts, we can assess which performs best for a given execution policy and computing environment.

Layout 1: Entries in each matrix are grouped together with each each having row major ordering; i.e.,

$$A = [A^0_{00}, A^0_{01}, A^0_{02}, A^0_{10}, A^0_{11}, A^0_{12}, A^0_{20}, A^0_{21}, A^0_{22},$$
$$A^1_{00}, A^1_{01}, A^1_{02}, A^1_{10}, A^1_{11}, A^1_{12}, A^1_{20}, A^1_{21}, A^1_{22}];$$

Layout 2: Matrix entries are first ordered by matrix index, then by column index, and finally by row index; i.e.,

$$A = [A^0_{00}, A^1_{00}, A^0_{01}, A^1_{01}, A^0_{02}, A^1_{02}, A^0_{10}, A^1_{10}, A^0_{11},$$
$$A^1_{11}, A^0_{12}, A^1_{12}, A^0_{20}, A^1_{20}, A^0_{21}, A^1_{21}, A^0_{22}, A^1_{22}];$$

Permuted Layouts

Next, we show how to construct the two data layouts using `RAJA::View` and `RAJA::Layout` objects. For more details on these RAJA concepts, please refer to *View and Layout*.

Layout 1 is constructed as follows:

```
// Dimensions of matrices
  const int N_c = 3;
  const int N_r = 3;

  std::array<RAJA::idx_t, 3> perm1 {{0, 1, 2}};
  auto layout1 =
      RAJA::make_permuted_layout( {{N, N_r, N_c}}, perm1 );
//
// RAJA::Layout objects may be templated on dimension, argument type, and
// index with unit stride. Here, the column index has unit stride (argument 2).
//
  RAJA::View<double, RAJA::Layout<3, Index_type, 2>> Aview(A, layout1);
  RAJA::View<double, RAJA::Layout<3, Index_type, 2>> Bview(B, layout1);
  RAJA::View<double, RAJA::Layout<3, Index_type, 2>> Cview(C, layout1);
```

The first argument to `RAJA::make_permuted_layout` is a C++ array whose entries correspond to the size of each array dimension; i.e., we have 'N' $N_r \times N_c$ matrices. The second argument describes the striding order of the array dimensions. Note that since this case follows the default RAJA ordering convention (see *View and Layout*), we use the identity permutation '(0,1,2)'.

For each matrix, the column index (index 2) has unit stride and the row index (index 1) has stride 3 (number of columns). The matrix index (index 0) has stride 9 ($N_c \times N_r$).

Layout 2 is constructed similarly:

```
  std::array<RAJA::idx_t, 3> perm2 {{1, 2, 0}};
  auto layout2 =
      RAJA::make_permuted_layout( {{N, N_r, N_c}}, perm2 );

  RAJA::View<double, RAJA::Layout<3, Index_type, 0>> Aview2(A2, layout2);
  RAJA::View<double, RAJA::Layout<3, Index_type, 0>> Bview2(B2, layout2);
  RAJA::View<double, RAJA::Layout<3, Index_type, 0>> Cview2(C2, layout2);
```

Here, the first argument to `RAJA::make_permuted_layout` is the same as in Layout 1 since we have the same number of matrices, matrix dimensions and we will use the same indexing scheme to access the matrix entries. However, the permutation we use is '(1,2,0)'.

This makes the matrix index (index 0) have unit stride, the column index (index 2) for each matrix has stride N, which is the number of matrices, and the row index (index 1) has stride $N \times N_c$.

Example Code

A complete working example that runs the batched matrix-multiplication computation for both layouts and various RAJA execution policies is located in the file `RAJA/examples/offset-layout.cpp`. It compares the execution run times of the two layouts using three RAJA back-ends (Sequential, OpenMP, and CUDA). The code example below shows the OpenMP version:

```
    RAJA::forall<RAJA::omp_parallel_for_exec>(
        RAJA::RangeSegment(0, N), [=](Index_type e) {

          Cview(e, 0, 0) = Aview(e, 0, 0) * Bview(e, 0, 0)
                         + Aview(e, 0, 1) * Bview(e, 1, 0)
                         + Aview(e, 0, 2) * Bview(e, 2, 0);
          Cview(e, 0, 1) = Aview(e, 0, 0) * Bview(e, 0, 1)
                         + Aview(e, 0, 1) * Bview(e, 1, 1)
                         + Aview(e, 0, 2) * Bview(e, 2, 1);
          Cview(e, 0, 2) = Aview(e, 0, 0) * Bview(e, 0, 2)
                         + Aview(e, 0, 1) * Bview(e, 1, 2)
                         + Aview(e, 0, 2) * Bview(e, 2, 2);

          Cview(e, 1, 0) = Aview(e, 1, 0) * Bview(e, 0, 0)
                         + Aview(e, 1, 1) * Bview(e, 1, 0)
                         + Aview(e, 1, 2) * Bview(e, 2, 0);
          Cview(e, 1, 1) = Aview(e, 1, 0) * Bview(e, 0, 1)
                         + Aview(e, 1, 1) * Bview(e, 1, 1)
                         + Aview(e, 1, 2) * Bview(e, 2, 1);
          Cview(e, 1, 2) = Aview(e, 1, 0) * Bview(e, 0, 2)
                         + Aview(e, 1, 1) * Bview(e, 1, 2)
                         + Aview(e, 1, 2) * Bview(e, 2, 2);

          Cview(e, 2, 0) = Aview(e, 2, 0) * Bview(e, 0, 0)
                         + Aview(e, 2, 1) * Bview(e, 1, 0)
                         + Aview(e, 2, 2) * Bview(e, 2, 0);
          Cview(e, 2, 1) = Aview(e, 2, 0) * Bview(e, 0, 1)
                         + Aview(e, 2, 1) * Bview(e, 1, 1)
                         + Aview(e, 2, 2) * Bview(e, 2, 1);
          Cview(e, 2, 2) = Aview(e, 2, 0) * Bview(e, 0, 2)
                         + Aview(e, 2, 1) * Bview(e, 1, 2)
                         + Aview(e, 2, 2) * Bview(e, 2, 2);

        });
```

All versions use the exact same lambda loop body showing that data orderings using RAJA can be altered similarly to execution policies without modifying application source code directly.

Stencil Computations (View Offsets)

Key RAJA features shown in the following example:

- `RAJA::Kernel` loop execution template
- RAJA kernel execution policies
- `RAJA::View` multi-dimensional data access
- `RAJA:make_offset_layout` method to apply index offsets

This example applies a five-cell stencil sum to the interior cells of a two-dimensional square lattice and stores the resulting sums in a second lattice of equal size. The five-cell stencil accumulates values from each interior cell and its

four neighbors. We use `RAJA::View` and `RAJA::Layout` constructs to simplify the multi-dimensional indexing so that we can write the stencil operation as follows:

```
output(row, col) = input(row, col) +
                   input(row - 1, col) + input(row + 1, col) +
                   input(row, col - 1) + input(row, col + 1)
```

A lattice is assumed to have $N_r \times N_c$ interior cells with unit values surrounded by a halo of cells containing zero values for a total dimension of $(N_r + 2) \times (N_c + 2)$. For example, when $N_r = N_c = 3$, the input lattice and values are:

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

After applying the stencil, the output lattice and values are:

0	0	0	0	0
0	3	4	3	0
0	4	5	4	0
0	3	4	3	0
0	0	0	0	0

For this $(N_r + 2) \times (N_c + 2)$ lattice case, here is our (row, col) indexing scheme.

(-1, 3)	(0, 3)	(1, 3)	(2, 3)	(3, 3)
(-1, 2)	(0, 2)	(1, 2)	(2, 2)	(3, 2)
(-1, 1)	(0, 1)	(1, 1)	(2, 1)	(3, 1)
(-1, 0)	(0, 0)	(1, 0)	(2, 0)	(3, 0)
(-1, -1)	(0, -1)	(1, -1)	(2, -1)	(3, -1)

Notably $[0, N_r) \times [0, N_c)$ corresponds to the interior index range over which we apply the stencil, and $[-1, N_r] \times [-1, N_c]$ is the full lattice index range.

RAJA Offset Layouts

We use the `RAJA::make_offset_layout` method to construct a `RAJA::OffsetLayout` object that defines our two-dimensional indexing scheme. Then, we create two `RAJA::View` objects for each of the input and output lattice arrays.

```
  const int DIM = 2;

  RAJA::OffsetLayout<DIM> layout =
      RAJA::make_offset_layout<DIM>({{-1, -1}}, {{N_r, N_c}});

  RAJA::View<int, RAJA::OffsetLayout<DIM>> input_latticeView(input_lattice, layout);
  RAJA::View<int, RAJA::OffsetLayout<DIM>> output_latticeView(output_lattice, layout);
```

Here, the row index range is $[-1, N_r]$, and the column index range is $[-1, N_c]$. The first argument to each call to the `RAJA::View` constructor is a pointer to an array that holds the data for the view; we assume the arrays are properly allocated before these calls.

The offset layout mechanics of RAJA allow us to write loops over data arrays using non-zero based indexing and without having to manually compute the proper offsets into the arrays. For more details on the `RAJA::View` and `RAJA::Layout` concepts we use in this example, please refer to *View and Layout*.

RAJA Kernel Implementation

For the RAJA implementations of the example computation, we use two `RAJA::RangeSegment` objects to define the row and column iteration spaces for the interior cells:

```
RAJA::RangeSegment col_range(0, N_r);
RAJA::RangeSegment row_range(0, N_c);
```

Here, is an implementation using `RAJA::kernel` multi-dimensional loop execution with a sequential execution policy.

```
using NESTED_EXEC_POL1 =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::seq_exec,     // row
      RAJA::statement::For<0, RAJA::seq_exec,   // col
        RAJA::statement::Lambda<0>
      >
    >
  >;

RAJA::kernel<NESTED_EXEC_POL1>(RAJA::make_tuple(col_range, row_range),
                               [=](int col, int row) {

                                 output_latticeView(row, col) =
                                     input_latticeView(row, col)
                                     + input_latticeView(row - 1, col)
                                     + input_latticeView(row + 1, col)
                                     + input_latticeView(row, col - 1)
                                     + input_latticeView(row, col + 1);
                               });
```

Since the stencil operation is data parallel, any parallel execution policy may be used. The file `RAJA/examples/tut_offset-layout.cpp` contains a complete working example code with various parallel implementations. For more details about `RAJA::kernel` concepts, please see *Complex Loops (RAJA::kernel)*.

Tiled Matrix Transpose

Key RAJA features shown in this example:

- `RAJA::kernel` usage with multiple lambdas
- `RAJA::statement::Tile` policy type

In this example, we compute the transpose of an input matrix A of size $N_r \times N_c$ and store the result in a second matrix At of size $N_c \times N_r$.

We compute the matrix transpose using a tiling algorithm, which iterates over tiles of the matrix A and performs a transpose copy of a tile without explicitly storing the tile. The algorithm is expressed as a collection of outer and inner for-loops. Iterations of the inner loop will transpose tile entries; while outer loops will iterate over the tiles needed to compute the transpose.

We start with a non-RAJA C++ implementation, where we choose tile dimensions smaller than the matrix dimensions. Note that we do not assume that tiles divide evenly the number of rows and and columns of the matrix. However, we do assume square tiles.

```
  const int N_r = 56;
  const int N_c = 75;
  const int TILE_DIM = 16;
```

Next, we calculate the number of tiles needed to carryout the transpose.

```
  const int outer_Dimc = (N_c - 1) / TILE_DIM + 1;
  const int outer_Dimr = (N_r - 1) / TILE_DIM + 1;
```

Then, the C++ implementation may look like the following:

```
  //
  // (0) Outer loops to iterate over tiles
  //
  for (int by = 0; by < outer_Dimr; ++by) {
    for (int bx = 0; bx < outer_Dimc; ++bx) {
      //
      // (1) Loops to iterate over tile entries
      //
      for (int ty = 0; ty < TILE_DIM; ++ty) {
        for (int tx = 0; tx < TILE_DIM; ++tx) {

          int col = bx * TILE_DIM + tx;  // Matrix column index
          int row = by * TILE_DIM + ty;  // Matrix row index

          // Bounds check
          if (row < N_r && col < N_c) {
            Atview(col, row) = Aview(row, col);
          }
        }
      }

    }
```

Note that we include a bounds check in the code to avoid indexing out of bounds when the tile sizes do not divide the matrix dimensions evenly.

RAJA::kernel Variant

For the `RAJA::kernel` variant, we use `RAJA::statement::Tile` types for the outer loop tiling, with `RAJA::statement::tile_fixed` parameters which identify the tile dimensions. The `RAJA::statement::Tile` types compute the number of tiles needed to iterate over all matrix entries in each dimension and generate iteration index values within the bounds of the associated iteration space. The complete sequential RAJA variant is given below:

```
  using KERNEL_EXEC_POL =
    RAJA::KernelPolicy<
      RAJA::statement::Tile<1, RAJA::statement::tile_fixed<TILE_DIM>, RAJA::seq_exec,
        RAJA::statement::Tile<0, RAJA::statement::tile_fixed<TILE_DIM>, RAJA::seq_
↪exec,
          RAJA::statement::For<1, RAJA::seq_exec,
            RAJA::statement::For<0, RAJA::seq_exec,
```

(continues on next page)

(continued from previous page)

```
              RAJA::statement::Lambda<0>
            >
          >
        >
      >
    >;

  RAJA::kernel<KERNEL_EXEC_POL>( RAJA::make_tuple(col_Range, row_Range),
    [=](int col, int row) {
      Atview(col, row) = Aview(row, col);
  });
```

The file `RAJA/examples/tut_tiled-matrix-transpose.cpp` contains the complete working example code for the examples described in this section, including OpenMP and CUDA variants.

A more advanced version using RAJA local arrays for CPU cache blocking and using GPU shared memory is discussed in *Matrix Transpose with Local Array*.

Matrix Transpose with Local Array

This section extends discussion in *Tiled Matrix Transpose*, where only loop tiling is considered. Here, we combine loop tiling with `RAJA::LocalArray` objects which enable CPU stack-allocated arrays, and GPU thread local and shared memory to be used within kernels. For more information about `RAJA::LocalArray`, please see *Local Array*.

Key RAJA features shown in this example:

- `RAJA::kernel_param` method with multiple lambda expressions
- `RAJA::statement::Tile` type
- `RAJA::statement::ForICount` type
- `RAJA::LocalArray`

As in *Tiled Matrix Transpose*, this example computes the transpose of an input matrix A of size $N_r \times N_c$ and stores the result in a second matrix At of size $N_c \times N_r$. The operation uses a local memory tiling algorithm. The algorithm tiles the outer loops and iterates over tiles in inner loops. The algorithm first loads input matrix entries into a local stack-allocated two-dimensional array for a tile, and then reads from the tile swapping the row and column indices to generate the output matrix.

We start with a non-RAJA C++ implementation to show the algorithm pattern. We choose tile dimensions smaller than the dimensions of the matrix and note that it is not necessary for the tile dimensions to divide evenly the number of rows and columns in the matrix A.

```
const int N_r = 267;
const int N_c = 251;

const int TILE_DIM = 16;
```

Next, we calculate the number of tiles needed to perform the transpose.

```
const int outer_Dimc = (N_c - 1) / TILE_DIM + 1;
const int outer_Dimr = (N_r - 1) / TILE_DIM + 1;
```

The complete C++ implementation of the tiled transpose operation using local memory is:

```
//
// (0) Outer loops to iterate over tiles
//
for (int by = 0; by < outer_Dimr; ++by) {
  for (int bx = 0; bx < outer_Dimc; ++bx) {

    // Stack-allocated local array for data on a tile
    int Tile[TILE_DIM][TILE_DIM];

    //
    // (1) Inner loops to read input matrix tile data into the array
    //
    //     Note: loops are ordered so that input matrix data access
    //           is stride-1.
    //
    for (int ty = 0; ty < TILE_DIM; ++ty) {
      for (int tx = 0; tx < TILE_DIM; ++tx) {

        int col = bx * TILE_DIM + tx;  // Matrix column index
        int row = by * TILE_DIM + ty;  // Matrix row index

        // Bounds check
        if (row < N_r && col < N_c) {
          Tile[ty][tx] = Aview(row, col);
        }
      }
    }

    //
    // (2) Inner loops to write array data into output array tile
    //
    //     Note: loop order is swapped from above so that output matrix
    //           data access is stride-1.
    //
    for (int tx = 0; tx < TILE_DIM; ++tx) {
      for (int ty = 0; ty < TILE_DIM; ++ty) {

        int col = bx * TILE_DIM + tx;  // Matrix column index
        int row = by * TILE_DIM + ty;  // Matrix row index

        // Bounds check
        if (row < N_r && col < N_c) {
          Atview(col, row) = Tile[ty][tx];
        }
      }
    }

  }
}
```

Note:

- To prevent indexing out of bounds, when the tile dimensions do not divide evenly the matrix dimensions, we use a bounds check in the inner loops.
- For efficiency, we order the inner loops so that reading from the input matrix and writing to the output matrix both use stride-1 data access.

RAJA::kernel Version

RAJA provides mechanisms to tile loops and use stack-allocated local arrays in kernels so that algorithm patterns like we just described can be implemented with RAJA. A `RAJA::LocalArray` type specifies an object whose memory is created inside a kernel using a `RAJA::statement` type in a RAJA kernel execution policy. The local array data is only usable within the kernel. See *Local Array* for more information.

`RAJA::kernel` methods also support loop tiling statements which determine the number of tiles needed to perform an operation based on tile size and extent of the corresponding iteration space. Moreover, lambda expressions for the kernel will not be invoked for iterations outside the bounds of an iteration space when tile dimensions do not divide evenly the size of the iteration space; thus, no conditional checks on loop bounds are needed inside inner loops.

For the RAJA version of the matrix transpose kernel above, we define the type of the `RAJA::LocalArray` used for matrix entries in a tile:

```
  using TILE_MEM =
    RAJA::LocalArray<int, RAJA::Perm<0, 1>, RAJA::SizeList<TILE_DIM, TILE_DIM>>;
```

The template parameters that define the type are: array data type, data stride permutation for the array indices (here the identity permutation is given, so the default RAJA conventions apply; i.e., the rightmost array index will be stride-1), and the array dimensions.

Here is the complete RAJA implementation for sequential CPU execution with kernel execution policy and kernel:

```
  using SEQ_EXEC_POL =
    RAJA::KernelPolicy<
      RAJA::statement::Tile<1, RAJA::statement::tile_fixed<TILE_DIM>, RAJA::loop_exec,
        RAJA::statement::Tile<0, RAJA::statement::tile_fixed<TILE_DIM>, RAJA::loop_
↪exec,

          RAJA::statement::InitLocalMem<RAJA::cpu_tile_mem, RAJA::ParamList<2>,

          RAJA::statement::ForICount<1, RAJA::statement::Param<0>, RAJA::loop_exec,
            RAJA::statement::ForICount<0, RAJA::statement::Param<1>, RAJA::loop_exec,
              RAJA::statement::Lambda<0>
            >
          >,

          RAJA::statement::ForICount<0, RAJA::statement::Param<1>, RAJA::loop_exec,
            RAJA::statement::ForICount<1, RAJA::statement::Param<0>, RAJA::loop_exec,
              RAJA::statement::Lambda<1>
            >
          >

          >
        >
      >
    >;

  RAJA::kernel_param<SEQ_EXEC_POL>( RAJA::make_tuple(RAJA::RangeSegment(0, N_c),
                                                     RAJA::RangeSegment(0, N_r)),

    RAJA::make_tuple((int)0, (int)0, RAJA_Tile),

    [=](int col, int row, int tx, int ty, TILE_MEM &RAJA_Tile) {
```

(continues on next page)

(continued from previous page)

```
      RAJA_Tile(ty, tx) = Aview(row, col);
    },

    [=](int col, int row, int tx, int ty, TILE_MEM &RAJA_Tile) {
      Atview(col, row) = RAJA_Tile(ty, tx);

  });
```

The `RAJA::statement::Tile` types at the start of the execution policy define tiling of the outer 'row' (iteration space tuple index '1') and 'col' (iteration space tuple index '0') loops, including tile sizes (`RAJA::statement::tile_fixed` types) and loop execution policies. Next, the `RAJA::statement::InitLocalMem` type initializes the local stack array based on the memory policy type (`RAJA::cpu_tile_mem`). The `RAJA::ParamList<2>` parameter indicates that the local array object is associated with position '2' in the parameter tuple argument passed to the `RAJA::kernel_param` method. Finally, we have two sets of nested inner loops for reading the input matrix entries into the local array and writing them out to the output matrix transpose. The inner bodies of each of these loop nests are identified by lambda expression arguments '0' and '1', respectively.

A couple of notes about the nested inner loops are worth emphasizing. First, the loops use `RAJA::statement::ForICount` types rather than `RAJA::statement::For` types that we have seen in earlier `RAJA::kernel` nested loop examples. The `RAJA::statement::ForICount` type generates local tile indices that are passed to lambda loop body expressions. As the observant reader will observe, there is no local tile index computation needed in the lambdas for the RAJA version of the kernel as a result. The first integer template parameter for each `RAJA::statement::ForICount` type indicates the item in the iteration space tuple passed to the `RAJA::kernel_param` method to which it applies; this is similar to `RAJA::statement::For` usage. The second template parameter for each `RAJA::statement::ForICount` type indicates the position in the parameter tuple passed to the `RAJA::kernel_param` method that will hold the associated local tile index. The loop execution policy template argument that follows works the same as in `RAJA::statement::For` usage. For more detailed discussion of RAJA loop tiling statement types, please see *Loop Tiling*.

Now that we described the execution policy in some detail, let's pull everything together by briefly walking though the call to the `RAJA::kernel_param` method. The first argument is a tuple of iteration spaces that define the iteration pattern for each level in the loop nest. Again, the first integer parameters given to the `RAJA::statement::Tile` and `RAJA::statement::ForICount` types identify the tuple entry they apply to. The second argument is a tuple of data parameters that will hold the local tile indices and `RAJA::LocalArray` tile memory. The tuple entries are associated with various statements in the execution policy as we described earlier. Next, two lambda expression arguments are passed to the `RAJA::kernel_param` method for reading and writing the input and output matrix entries, respectively.

Note that each lambda expression takes five arguments. The first two are the matrix column and row indices associated with the iteration space tuple. The next three arguments correspond to the parameter tuple entries. The first two of these are the local tile indices used to access entries in the `RAJA::LocalArray` object memory. The last argument is a reference to the `RAJA::LocalArray` object itself.

The file `RAJA/examples/tut_matrix-transpose-local-array.cpp` contains the complete working example code for the examples described in this section along with OpenMP and CUDA variants.

3.5 Using RAJA in Your Application

Using RAJA in an application requires two things: ensuring the header files are visible, and linking against the RAJA library.

3.5.1 CMake Configuration File

As part of the RAJA installation, we provide a `RAJA-config.cmake` file. If your application uses CMake, this can be used with CMake's find_package capability to import RAJA into your CMake project.

To use the configuration file, you can add the following command to your CMake project:

```
find_package(RAJA)
```

Then, pass the path of RAJA to CMake when you configure your code:

```
cmake -DRAJA_DIR=<path-to-raja>/share/raja/cmake
```

The `RAJA-config.cmake` file provides the following variables:

Variable	Default
`RAJA_INCLUDE_DIR`	Include directory for RAJA headers.
`RAJA_LIB_DIR`	Library directory for RAJA.
`RAJA_COMPILE_FLAGS`	C++ flags used to compile RAJA.
`RAJA_NVCC_FLAGS`	CUDA flags used to compile RAJA.

It also provides the `RAJA` target, that can be used natively by CMake to add a dependency on RAJA. For example:

```
add_executable(my-app.exe
               my-app.cpp)

target_link_libraries(my-app.exe RAJA)

target_include_directories(my-app.exe ${RAJA_INCLUDE_DIR}
```

3.6 Build Configuration Options

RAJA uses BLT, a CMake-based build system. In *Getting Started With RAJA*, we described how to run CMake to configure RAJA with its default option settings. In this section, we describe all RAJA configuration options, their defaults, and how to enable or disable features.

3.6.1 Setting Options

The RAJA configuration can be set using standard CMake variables along with BLT and RAJA-specific variables. For example, to make a release build with some system default GNU compiler and then install the RAJA header files and libraries in a specific directory location, you could do the following in the top-level RAJA directory:

```
$ mkdir build-gnu-release
$ cd build-gnu-release
$ cmake \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_C_COMPILER=gcc \
  -DCMAKE_CXX_COMPILER=g++ \
  -DCMAKE_INSTALL_PREFIX=../install-gnu-release ../
$ make
$ make install
```

Following CMake conventions, RAJA supports three build types: `Release`, `RelWithDebInfo`, and `Debug`. Similar to other CMake systems, when you choose a build type that includes debug information, you do not have to specify the '-g' compiler flag to generate debugging symbols.

All RAJA options are set like standard CMake variables. For example, to enable RAJA OpenMP functionality, pass the following argument to cmake:

```
-DENABLE_OPENMP=On
```

All RAJA settings for default options, compilers, flags for optimization, etc. can be found in files in the `RAJA/cmake` directory. Next, we summarize the available options and their defaults

3.6.2 Available Options and Defaults

RAJA uses a variety of custom variables to control how it is compiled. Many of these are used internally to control RAJA compilation and do not need to be set by users. Others can be used to enable or disable certain RAJA features. Most variables get translated to compiler directives and definitions in the RAJA `config.hpp` file that is generated when CMake runs. The `config.hpp` header file is included in other RAJA headers as needed so all options propagate consistently through the build process for all of the code. Each RAJA variable has a special prefix to distinguish it as being specific to RAJA; i.e., it is not a BLT variable or a standard CMake variable.

The following tables describe which variables set RAJA options and and their default settings:

- **Examples, tests, warnings, etc.**

  Variables that control whether RAJA tests and examples are built when the library is compiled are:

Variable	Default
ENABLE_TESTS	On
ENABLE_EXAMPLES	On

  RAJA can also be configured to build with compiler warnings reported as errors, which may be useful when using RAJA in an application:

Variable	Default
ENABLE_WARNINGS_AS_ERRORS	Off

- **Programming models and compilers**

  Variables that control which RAJA programming model back-ends are enabled are (names are descriptive of what they enable):

Variable	Default
ENABLE_OPENMP	On
ENABLE_TARGET_OPENMP	Off
ENABLE_CUDA	Off
ENABLE_TBB	Off

  Other compilation options are available via the following:

Variable	Default
ENABLE_CLANG_CUDA	Off
ENABLE_CUB	On (when CUDA enabled)

Turning the 'ENABLE_CLANG_CUDA' variable on will build CUDA code with the native support in the Clang compiler. When using it, the 'ENABLE_CUDA' variable must also be turned on.

The 'ENABLE_CUB' variable is used to enable NVIDIA CUB library support for RAJA CUDA scans. Since the CUB library is included in RAJA as a Git submodule, users should not have to set this in most scenarios.

Note: When using the NVIDIA nvcc compiler for RAJA CUDA functionality, the variable 'RAJA_NVCC_FLAGS' should be used to pass flags to nvcc.

- **Data types, sizes, alignment, etc.**

  RAJA provides type aliases that can be used to parameterize floating point types in applications, which makes it easy to switch between types.

  The following variables are used to set the data type for the type alias `RAJA::Real_type`:

Variable	Default
RAJA_USE_DOUBLE	On
RAJA_USE_FLOAT	Off

  Similarly, the 'RAJA::Complex_type' can be enabled to support complex numbers if needed:

Variable	Default
RAJA_USE_COMPLEX	Off

  When turned on, the RAJA Complex_type is 'std::complex<Real_type>'.

  There are several variables to control the definition of the RAJA floating-point data pointer type `RAJA::Real_ptr`. The base data type is always `Real_type`. When RAJA is compiled for CPU execution only, the defaults are:

Variable	Default
RAJA_USE_BARE_PTR	Off
RAJA_USE_RESTRICT_PTR	On
RAJA_USE_RESTRICT_ALIGNED_PTR	Off
RAJA_USE_PTR_CLASS	Off

  When RAJA is compiled with CUDA enabled, the defaults are:

Variable	Default
RAJA_USE_BARE_PTR	On
RAJA_USE_RESTRICT_PTR	Off
RAJA_USE_RESTRICT_ALIGNED_PTR	Off
RAJA_USE_PTR_CLASS	Off

  The meaning of these variables is:

Variable	Meaning
RAJA_USE_BARE_PTR	Use standard C-style pointer
RAJA_USE_RESTRICT_PTR	Use C-style pointer with restrict qualifier
RAJA_USE_RESTRICT_ALIGNED_PTR	Use C-style pointer with restrict qualifier and alignment attribute (see RAJA_DATA_ALIGN below)
RAJA_USE_PTR_CLASS	Use pointer class with overloaded *[]* operator that applies restrict and alignment intrinsics. This is useful when a compiler does not support attributes in a typedef.

RAJA internally uses parameters to define platform-specific constants for index ranges and data alignment. The variables that control these are:

Variable	Default
RAJA_RANGE_ALIGN	4
RAJA_RANGE_MIN_LENGTH	32
RAJA_DATA_ALIGN	64

What these variables mean:

Variable	Meaning
RAJA_RANGE_ALIGN	Constrain alignment of begin/end indices of range segments generated by index set builder methods; i.e., begin and end indices of such segments will be multiples of this value.
RAJA_RANGE_MIN_LENGTH	Set minimum length of range segments generated by index set builder methods. This should be an integer multiple of RAJA_RANGE_ALIGN.
RAJA_DATA_ALIGN	Specifies data alignment used in intrinsics and typedefs; units of **bytes**.

For details on the options in this section are used, please see the header file `RAJA/include/RAJA/util/types.hpp`.

- **Timer Options**

  RAJA provides a simple portable timer class that is used in RAJA example codes to determine execution timing and can be used in other apps as well. This timer can use any of three internal timers depending on your preferences, and one should be selected by setting the 'RAJA_TIMER' variable. If the 'RAJA_CALIPER' variable is turned on (off by default), the timer will also offer caliper-based region annotations.

Variable	Values
RAJA_TIMER	chrono (default) gettime clock

What these variables mean:

Value	Meaning
chrono	Use the std::chrono library from the C++ standard library
gettime	Use *timespec* from the C standard library time.h file
clock	Use *clock_t* from time.h

- **Other RAJA Features**

RAJA contains some features that are used mainly for development or may not be of general interest to RAJA users. These are turned off be default. They are described here for reference and completeness.

Variable	Meaning
ENABLE_CHAI	Enable/disable RAJA internal support for CHAI
ENABLE_FT	Enable/disable RAJA experimental loop-level fault-tolerance mechanism
RAJA_REPORT_FT	Enable/disable a report of fault- tolerance enabled run (e.g., number of faults detected, recovered from, recovery overhead, etc.)

3.6.3 RAJA Host-Config Files

The `RAJA/host-configs` directory contains subdirectories with files that define configurations for various platforms and compilers at LLNL. These serve as examples of *CMake cache files* that can be passed to CMake using the '-C' option. This option initializes the CMake cache with the configuration specified in each file. For examples of how they are used for specific CMake configurations, see the build scripts in `RAJA/scripts` subdirectories that can be used to drive the RAJA 'host-config' files.

3.7 Plugins

RAJA provides a plugin mechanism to support optional components that provide additional functionality to make writing applications easier. Currently, there is only one RAJA plugin that we support, CHAI.

3.7.1 CHAI

RAJA provides abstractions for parallel execution, but does not support a memory model for managing data in heterogeneous memory spaces. CHAI is an array abstraction that can be used to copy data transparently from one memory space to another as needed to run a RAJA-based kernel. The data can be accessed inside any RAJA kernel, and regardless of where that kernel executes, CHAI will make the data available.

To build RAJA with CHAI integration, you need to download and install CHAI. Please see the CHAI project for details.

After CHAI is installed, RAJA can be configured to use it by passing two additional arguments to CMake:

```
$ cmake -DRAJA_ENABLE_CHAI=On -Dchai_DIR=/path/to/chai
```

After RAJA has been built with CHAI support enabled, applications can use `chai::ManangedArray` objects to access data inside RAJA kernels; for example:

```
chai::ManagedArray<float> array(1000);

RAJA::forall<RAJA::cuda_exec<16> >(0, 1000, [=] __device__ (int i) {
    array[i] = i * 2.0f;
});

RAJA::forall<RAJA::seq_exec>(0, 1000, [=] (int i) {
  std::cout << "array[" << i << "]  is " << array[i] << std::endl;
});
```

Here, the data held by `array` is allocated on the host CPU. Then, it is initialized on a CUDA GPU device. CHAI sees that the data lives on the CPU and is needed in a GPU device data environment. So it copies the data from CPU to GPU, making it available for access in the first RAJA kernel. Next, it is printed in the second kernel which runs on the CPU. So CHAI copies the data back to the host CPU. All necessary data copies are done transparently on demand as needed for each kernel.

3.8 Contributing to RAJA

This section is intended for folks who want to contribute new features or bugfixes to RAJA. It assumes you are familiar with Git and GitHub. It describes what a good pull request (PR) looks like, and the tests that your PR must pass before it can be merged into RAJA.

3.8.1 Forking RAJA

If you aren't a RAJA developer at LLNL, then you won't have permission to push new branches to the repository. First, you should create a fork of the repo. This will create a copy of the RAJA repository that you own, and will ensure you can push your changes to GitHub and create pull requests.

3.8.2 Developing a New Feature

New features should be based on the RAJA `develop` branch. When you want to create a new feature, first ensure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

Then, create a new branch to develop your feature on:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to develop your feature on this branch pushing changes with reasonably-sized atomic commits, and add tests that will exercise your new code. If you are creating new methods or classes, please add Doxygen documentation.

Once your feature is complete and your tests are passing, you can push your branch to GitHub and create a PR.

3.8.3 Developing a Bug Fix

First, check if the change you want to make has been addressed in the RAJA `develop` branch. If so, we suggest you either start using the `develop` branch, or temporarily apply the fix to whichever version of RAJA you are using.

Assuming there is an unsolved bug, first make sure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

Then create a new branch for your bugfix:

```
$ git checkout -b bugfix/<name-of-bug>
```

First, add a test that reproduces the bug you have found. Then develop your bugfix as normal, and ensure to `make test` to check your changes actually fix the bug.

Once you are finished, you can push your branch to GitHub, then create a PR.

3.8.4 Creating a Pull Request

You can create a new PR here. GitHub has a good guide on PR basics if you want more information. Ensure that your PR base is the `develop` branch of RAJA.

Add a descriptive title explaining the bug you fixed or the feature you have added, and put a longer description of the changes you have made in the comment box.

Once your PR has been created, it will be run through our automated tests and also be reviewed by RAJA team members. Providing the branch passes both the tests and reviews, it will be merged into RAJA.

3.8.5 Tests

RAJA uses Travis CI for continuous integration tests. Our tests are automatically run against every new pull request, and passing all tests is a requirement for merging your PR. If you are developing a bugfix or a new feature, please add a test that checks the correctness of your new code. RAJA is used on a wide variety of systems with a number of configurations, and adding new tests helps ensure that all features work as expected across these environments.

All RAJA tests are in the `RAJA/test` directory and are split up by programming model back-end and feature.

3.9 RAJA License

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.

2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.