RAJA Documentation

Release 2022.03.1

LLNL

Dec 09, 2022

User Documentation

1 Background and Motivation 3

2 Git Repository and Issue Tracking 5

3 Communicating with the RAJA Team 7

4 RAJA User Documentation 9

5 RAJA Developer Documentation 11

6 RAJA Copyright and License Information 13

6.1 RAJA User Guide 13

6.2 Doxygen 140

6.3 RAJA Developer Guide 140

6.4 RAJA Copyright and License Information 151

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 existing algorithms and programming styles.
2. To achieve performance comparable to using common 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++ abstraction layer that helps address this performance portability challenge.

RAJA provides portable abstractions for simple and complex loops – as well reductions, scans, atomic operations, sorts, data layouts, views, and loop iteration spaces, as well as compile-time loop transformations. Features are continually growing as new use cases arise due to expanding user adoption.

RAJA uses standard C++11 – C++ is the programming language model of choice for many HPC applications. RAJA requirements and design are rooted in a decades of developer experience working on production mesh-based multiphysics applications. 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 application 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.

RAJA support for various execution back-ends is the result of collaborative development between the RAJA team and academic and industrial partners. Currently available execution back-ends include: sequential, SIMD, Threading Building Blocks (TBB), NVIDIA CUDA, OpenMP CPU multithreading and target offload, and AMD HIP. Sequential, CUDA, OpenMP CPU multithreading, and HIP execution are supported for all RAJA features. Sequential, OpenMP CPU multithreading, and CUDA are considered the most developed at this point as these have been our primary focus up to now. Those back-ends are used in a wide variety of production applications. OpenMP target offload and TBB back-ends do not support all RAJA features and should be considered experimental.

CHAPTER 2

Git Repository and Issue Tracking

The main interaction hub for RAJA is on GitHub

CHAPTER 3

Communicating with the RAJA Team

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.

The best way to communicate with us is via our email list: `raja-dev@llnl.gov`

You are also welcome to join our Google Group

A brief description of how to start a contribution to RAJA can be found in *Contributing to RAJA*.

CHAPTER 4

RAJA User Documentation

- *RAJA User Guide*
- RAJA Tutorials Repo
- Source Documentation

CHAPTER 5

RAJA Developer Documentation

- *RAJA Developer Guide*

CHAPTER 6

RAJA Copyright and License Information

Please see *RAJA Copyright and License Information*.

6.1 RAJA User 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*.

6.1.1 Getting Started With RAJA

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

Requirements

The primary requirement for using RAJA is a C++14 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 RAJA in its most basic form and use its simplest features:

- C++ compiler with C++14 support
- CMake version 3.14.5 or greater.

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 necessary RAJA dependencies as Git *submodules*. Current RAJA dependencies are:

- BLT build system
- Camp compiler agnostic metaprogramming library
- CUB CUDA utilities library
- rocPRIM HIP parallel primitives library

You probably don't need to know much about these other projects to start using RAJA. But, if you want to know more about them, 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 do not 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.

Build and Install

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

Building RAJA

RAJA uses CMake to configure a build. A "bare bones" configuration looks like:

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

Note:

- RAJA requires a minimum CMake version of 3.14.5.
- Builds must be *out-of-source*. RAJA does not allow building in the source directory, so you must create a build directory and run CMake in it.

When you run CMake, it will generate output about the build environment (compiler and version, options, etc.). Some RAJA features, like OpenMP support are enabled by default if, for example, the compiler supports OpenMP. These can be disabled if desired. For a summary of RAJA 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.,:

```
$ 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 (please see *Build Configuration Options*), you can run them after the build completes to check if everything is built properly.

  The easiest way to run the full set of RAJA tests is to type:

  ```
  $ make test
  ```

  in the build directory after the build completes.

  You can also run individual tests by invoking test executables directly. They will be located in the `test` subdirectory in the build space directory. RAJA tests use the Google Test framework, so you can also run tests via Google Test commands.

- RAJA also contains example and tutorial exercise programs you can run if you wish. Similar to the RAJA tests, the examples and exercises are built by default and can be disabled with CMake options (see *Build Configuration Options*). The source files for these are located in the `RAJA/examples` and `RAJA/exercises` directories, respectively. When built, the executables for the examples and exercises will be located in the `bin` subdirectory in the build space directory. Feel free to experiment by editing the source files and recompiling.

Note: You may use externally-supplied versions of the camp, CUB, and rocPRIM libraries with RAJA if you wish. To do so, pass the following options to CMake:

- External camp: -DEXTERNAL_CAMP_SOURCE_DIR=<camp dir name>
- External CUB: -DRAJA_ENABLE_EXTERNAL_CUB=On -DCUB_DIR=<CUB dir name>
- **External rocPRIM: -DRAJA_ENABLE_EXTERNAL_ROCPRIM=On** -DROCPRIM_DIR=<rocPRIM dir name>

GPU Builds, etc.

CUDA

To run RAJA code on NVIDIA GPUs, one typically must have a CUDA compiler installed on your system, in addition to a host code compiler. You may need to specify both when you run CMake. The host compiler is specified using the `CMAKE_CXX_COMPILER` CMake variable. The CUDA compiler is specified with the `CMAKE_CUDA_COMPILER` variable.

When using the NVIDIA nvcc compiler for RAJA CUDA functionality, the variables:

- CMAKE_CUDA_FLAGS_RELEASE
- CMAKE_CUDA_FLAGS_DEBUG
- CMAKE_CUDA_FLAGS_RELWITHDEBINFO

which corresponding to the standard CMake build types are used to pass flags to nvcc.

Note: When nvcc must pass options to the host compiler, the arguments can be included using these CMake variables. Host compiler options must be prepended with the *-Xcompiler* directive.

To set the CUDA compute architecture for the nvcc compiler, which should be chosen based on the NVIDIA GPU hardware you are using, you can use the `CUDA_ARCH` CMake variable. For example, the CMake option:

```
-DCUDA_ARCH=sm_60
```

will tell the compiler to use the *sm_60* SASS architecture in its second stage of compilation. It will pick the PTX architecture to use in the first stage of compilation that is suitable for the SASS architecture you specify.

Alternatively, you may specify the PTX and SASS architectures, using appropriate nvcc options in the `CMAKE_CUDA_FLAGS_*` variables.

Note: RAJA requires a minimum CUDA architecture level of ‘sm_35‘ to use all supported CUDA features. Mostly, the architecture level affects which RAJA CUDA atomic operations are available and how they are implemented inside RAJA. This is described in *Atomics*.

- If you do not specify a value for `CUDA_ARCH`, it will be set to *sm_35* by default and CMake will emit a status message indicatting this choice was made.
- If you give a `CUDA_ARCH` value less than *sm_35* (e.g., *sm_30*), CMake will report this and stop processing.

Also, RAJA relies on the CUB CUDA utilities library for some CUDA functionality. The CUB included in the CUDA toolkit is used by default if available. RAJA includes a CUB submodule that is used if it is not available. To use an external CUB install provide the following option to CMake: `-DRAJA_ENABLE_EXTERNAL_CUB=On -DCUB_DIR=<pat/to/cub>`.

Note: **It is important to note that the CUDA toolkit version of cub is required for compatibility with the CUDA toolkit version of thrust starting with CUDA toolkit version v11.0.0. So, if you build RAJA with CUDA version 11 or higher you must use the CUDA toolkit version of CUB to use Thrust and be compatible with libraries that use Thrust.

*It is important to note that the version of Googletest that is used in RAJA version v0.11.0 or newer requires CUDA version 9.2.x or newer when compiling with nvcc. Thus, if you build RAJA with CUDA enabled and want to also enable RAJA tests, you must use CUDA version 9.2.x or newer.

HIP

To run RAJA code on AMD GPUs, one typically uses the HIP compiler and tool chain (which can also be used to compile code for NVIDIA GPUs).

Note: RAJA requires version 3.5 or newer of the ROCm software stack to use the RAJA HIP back-end.

Also, RAJA relies on the rocPRIM HIP utilities library for some HIP functionality. The rocPRIM included in the ROCm install is used by default if available. RAJA includes a rocPRIM submodule that is used if it is not available. To use an external rocPRIM install provide the following option to CMake: `-DRAJA_ENABLE_EXTERNAL_ROCPRIM=On -DROCPRIM_DIR=<pat/to/rocPRIM>`.

Note: When using HIP and targeting NVIDIA GPUs RAJA uses CUB instead of rocPRIM. In this case you must use an external CUB install using the CMake variables described in the CUDA section.

OpenMP

To use OpenMP target offlad GPU execution, additional options may need to be passed to the compiler. The variable `OpenMP_CXX_FLAGS` is used for this. Option syntax follows the CMake *list* pattern. For example, to specify OpenMP target options for NVIDIA GPUs using a clang-based compiler, one may do something like:

```
cmake \
  ....
  -DOpenMP_CXX_FLAGS="-fopenmp;-fopenmp-targets=nvptx64-nvidia-cuda"
```

RAJA Example Build Configuration Files

The `RAJA/scripts` directory contains subdirectories with a variety of build scripts we use to build and test RAJA on various platforms with various compilers. These scripts pass files (*CMake cache files*) located in the `RAJA/host-configs` directory to CMake using the '-C' option. These files serve as useful examples of how to configure RAJA prior to compilation.

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.

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*.

6.1.2 RAJA Features

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

Elements of Loop Execution

In this section, we describe the basic elements of RAJA loop kernel execution. `RAJA::forall`, `RAJA::kernel`, and `RAJA::expt::launch` (aka *RAJA Teams*) template methods comprise the RAJA interface for loop execution. `RAJA::forall` methods execute simple, non-nested loops, `RAJA::kernel` methods support nested loops and other complex loop kernels and transformations, and `RAJA::expt::launch` creates an execution space in which algorithms are expressed in terms of nested loops using the `RAJA::expt::loop` method.

Note:

- The `forall` , and `kernel` methods are in the namespace `RAJA`, while `launch` is found under the RAJA namespace for experimental features `RAJA::expt`.
- 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, such as a contiguous range of loop indices, and
  - a single 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.
- The `RAJA::expt::launch` method is a template on both host and device policies to create an execution space for kernels. Since both host and device poilices are specified, the launch method can be used to select at run-time whether to run a kernel on the host or device. Algorithms are expressed inside the execution space as nested loops using `RAJA::loop` methods.
  - Hierarchical parallelism can be expressed using the thread and thread-team model with `RAJA::expt::loop` methods as found in programming models such as CUDA/HIP.

Various examples showing how to use `RAJA::forall`, `RAJA::kernel`, `RAJA::launch` 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 (i.e., non-nested) loops. For example, a C-style loop that adds two vectors, like this:

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

may be written using RAJA as:

```
RAJA::forall<exec_policy>(RAJA::RangeSegment(0, N), [=] (int 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 range segment (shown here), 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 enables 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 for that use case.

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 GPU kernels, generally) and would add excessive overhead for copying data into the lambda data environment when captured by value.

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

Note that we could write this by nesting `RAJA::forall` statements and it would work for some execution policy choices:

```
RAJA::forall<exec_policyN>(IN, [=] (int iN) {
  ...
     RAJA::forall<exec_policy0>(I0, [=] (int 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 may limit 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 without changing the source code, which breaks RAJA encapsulation.

Note: We do not recommend nesting ‘‘RAJA::forall‘‘ statements.

The RAJA *kernel* interface facilitates parallel execution and compile-time transformation of arbitrary loop nests and other complex loop structures. It can treat a complex loop structure as a single entity, which simplifies the ability to transform and apply different parallel execution patterns by changing the execution policy type and *not the kernel code*.

The loop above nest may be written using the RAJA kernel interface 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(RAJA::RangeSegment(0, NN), ..., RAJA::RangeSegment(0, N0),

  [=] (int iN, ... , int i0) {
     // inner loop body
  }

);
```

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 the `For` statement to the item in the iteration space tuple corresponding to that index,
- an execution policy type for the associated 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. One can think of the ‘<, >’ symbols enclosing the template parameter lists as being similar to the curly braces in C-style code.

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.

RAJA offers two types of lambda statements. The first as illustratated above, requires that each lambda expression passed to a `RAJA::kernel` method **must take an index argument for each iteration space in the tuple**. With this type of lambda statement, the entire iteration space must be active in a containing `For` construct. A compile time `static_assert` will be triggered if any of the arguments are undefined, indicating that something is not correct.

The second type of lambda statement, an extension of the first, takes additional template parameters which specify which iteration space indices are passed as lambda arguments. The result is that a kernel lambda only needs to accept iteration space index arguments that are used in the lambda body.

The kernel policy list with lambda arguments may be written as:

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

The template parameter `RAJA::Segs` is used to specify which elements in the segment tuple are used to pass arguments to a lambda. RAJA offers other types such as `RAJA::Offsets`, and `RAJA::Params` to identify offsets and parameters in segments and param tuples respectively to be used as lambda argumentsx. See *Matrix Multiplication (Nested Loops)* and *Matrix Transpose with Local Array* for detailed examples.

Note: Unless lambda arguments are specified in RAJA lambda statements, 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 a lambda, but they **all must appear** in the lambda argument list and **all must be in active loops** 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 with unused arguments.

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.

Note: In general, RAJA execution policies for `RAJA::forall` and `RAJA::kernel` are different. A summary of all RAJA execution policies that may be used with `RAJA::forall` or `RAJA::kernel` may be found in *Policies*.

Finally, a discussion of how to construct `RAJA::KernelPolicy` types and available `RAJA::statement` types can be found in *RAJA Kernel Execution Policies*.

Team based loops (RAJA::launch)

The *RAJA Teams* framework aims to unify thread/block based programming models such as CUDA/HIP/SYCL while maintaining portability on host backends (OpenMP, sequential). When using the `RAJA::kernel` interface, developers express all aspects of nested loop execution in the execution policy type on which the `RAJA::kernel` method is templated. In contrast, the `RAJA::launch` interface allows users to express nested loop execution in a manner that more closely reflects how one would write conventional nested C-style for-loop code. Additionally, *RAJA Teams* introduces run-time host or device selectable kernel execution. The main application of *RAJA Teams* is imperfectly nested loops. Using the `RAJA::expt::launch method` developers are provided with an execution space enabling them to express algorithms in terms of nested `RAJA::expt::loop` statements:

```
RAJA::expt::launch<launch_policy>(select_CPU_or_GPU)
RAJA::expt::Grid(RAJA::expt::Teams(NE), RAJA::expt::Threads(Q1D)),
[=] RAJA_HOST_DEVICE (RAJA::expt::Launch ctx) {

  RAJA::expt::loop<team_x> (ctx, RAJA::RangeSegment(0, teamRange), [&] (int bx) {

    RAJA_TEAM_SHARED double s_A[SHARE_MEM_SIZE];

    RAJA::expt::loop<thread_x> (ctx, RAJA::RangeSegment(0, threadRange), [&] (int tx)
↪{
      s_A[tx] = tx;
    });

      ctx.teamSync();

 )};

});
```

The underlying idea of *RAJA Teams* is to enable developers to express nested parallelism in terms of teams and threads. Similar to the CUDA programming model, development is done using a collection of threads, threads are grouped into teams. Using the `RAJA::expt::loop` methods iterations of the loop may be executed by threads or teams (depending on the execution policy). The launch context serves to synchronize threads within the same team. The *RAJA Teams* abstraction consist of three main concepts.

- *Launch Method*: creates an execution space in which developers may express their algorithm in terms of nested `RAJA::expt::loop` statements. The loops are then executed by threads or thread-teams. The method is templated on both a host and device execution space and enables run-time selection of the execution environment.
- *Resources*: holds a number of teams and threads (akin to CUDA blocks/threads).
- *Loops*: are used to express hierarchical parallelism. Work within a loop is mapped to either teams or threads. Team shared memory is available by using the `RAJA_TEAM_SHARED` macro. Team shared memory enables threads in a given team to share data. In practice, team policies are typically aliases for RAJA GPU block policies in the x,y,z dimensions (for example cuda_block_direct), while thread policies are aliases for RAJA GPU thread policies (for example cuda_thread_direct) x,y,z dimensions. On the host, teams and threads may be mapped to sequential loop execution or OpenMP threaded regions.

The team loop interface combines concepts from `RAJA::forall` and `RAJA::kernel`. Various policies from `RAJA::kernel` are compatible with the `RAJA Teams` framework.

MultiDimensional loops using Simple loop APIs (RAJA::CombiningAdapter)

A `RAJA::CombiningAdapter` object provides ways to run perfectly nested loops with simple loop APIs like `RAJA::forall` and `RAJA::WorkGroup` *WorkGroup*. To introduce the `RAJA ::CombiningAdapter` interface, consider a (N+1)-level C-style loop nest:

```
for (int iN = 0; iN < NN; ++iN) {
  ...
     for (int i0 = 0; i0 < N0; ++i0) {
```

(continues on next page)

(continued from previous page)

```
        \\ inner loop body
      }
}
```

We can use a `RAJA::CombiningAdapter` to combine the iteration spaces of the loops and pass the adapter to a `RAJA::forall` statement to execute them:

```
auto adapter = RAJA::make_CombingingAdapter(
    [=] (int iN, ..., int i0)) {
      \\ inner loop body
    }, IN, ..., I0);

RAJA::forall<exec_policy>(adapter.getRange(), adapter);
```

A `RAJA::CombiningAdapter` object is a template combining a loop body and iteration spaces. The maker function template takes a lambda expression for the loop body and an arbitrary number of segment arguments. It provides a flattened index space via the `getRange` method that can be passed as the iteration space to the simple loop API. The object itself can be passed into the loop API as the loop body. The object's call operator does the conversion of the flat single dimensional index into the multi-dimensional index space, calling the provided lambda with the appropriate indices.

Note: CombiningAdapter currently only supports `RAJA::RangeSegment` and `RAJA::TypedRangeSegment` segments.

Policies

This section describes RAJA policies for loop kernel execution, scans, sorts, 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 specifies additional information about the execution pattern, such as number of CUDA threads per thread block, whether execution is synchronous or asynchronous, etc.

As RAJA functionality evolves, 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`.
- All RAJA policies have a prefix indicating the back-end implementation that they use; e.g., `omp_` for OpenMP, `cuda_` for CUDA, etc.

RAJA Loop/Kernel Execution Policies

The following tables summarize RAJA policies for executing kernels. Please see notes below policy descriptions for additional usage details and caveats.

Sequential CPU Policies

For the sequential CPU back-end, RAJA provides policies that allow developers to have some control over the optimizations that compilers are allow to apply during code compilation.

Sequential/SIMD Execution Policies	Works with	Brief description
seq_exec	forall, kernel (For), scan, sort	Strictly sequential execution.
simd_exec	forall, kernel (For), scan	Try to force generation of SIMD instructions via compiler hints in RAJA's internal implementation.
loop_exec	forall, kernel (For), scan, sort	Allow the compiler to generate any optimizations that its heuristics deem beneficial according; i.e., no loop decorations (pragmas or intrinsics) in RAJA implementation.

OpenMP Parallel CPU Policies

For the OpenMP CPU multithreading back-end, RAJA has policies that can be used by themselves to execute kernels. In particular, they create an OpenMP parallel region and execute a kernel within it. To distinguish these in this discussion, we refer to these as **full policies**. These policies are provided to users for convenience in common use cases.

RAJA also provides other OpenMP policies, which we refer to as **partial policies**, since they need to be used in combination with other policies. Typically, they work by providing an *outer policy* and an *inner policy* as a template parameter to the outer policy. These give users flexibility to create more complex execution patterns.

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)' in your application, which allows one to change the number of threads at runtime.

The full policies are described in the following table. Partial policies are described in other tables below.

OpenMP CPU Full Policies	Works with	Brief description
omp_parallel_for_exec	forall, kernel (For), scan, sort	Same as applying 'omp parallel for' pragma
omp_parallel_for_static_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp parallel for schedule(static, ChunkSize)'
omp_parallel_for_dynamic_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp parallel for schedule(dynamic, ChunkSize)'
omp_parallel_for_guided_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp parallel for schedule(guided, ChunkSize)'
omp_parallel_for_runtime_exec	forall, kernel (For)	Same as applying 'omp parallel for schedule(runtime)'

Note: For the OpenMP scheduling policies above that take a `ChunkSize` parameter, the chunk size is optional. If not provided, the default chunk size that OpenMP applies will be used, which may be specific to the OpenMP implementation in use. For this case, the RAJA policy syntax is `omp_parallel_for_{static|dynamic|guided}_exec< >`, which will result in the OpenMP pragma `omp parallel for schedule({static|dynamic|guided})` being applied.

RAJA provides an (outer) OpenMP CPU policy to create a parallel region in which to execute a kernel. It requires an inner policy that defines how a kernel will execute in parallel inside the region.

OpenMP CPU Outer Policies	Works with	Brief description
omp_parallel_exec<InnerPolicy>	forall, kernel (For), scan	Creates OpenMP parallel region and requires an **InnerPolicy**. Same as applying 'omp parallel' pragma.

Finally, we summarize the inner policies that RAJA provides for OpenMP. These policies are passed to the RAJA `omp_parallel_exec` outer policy as a template argument as described above.

OpenMP CPU Inner Policies	Works with	Brief description
omp_for_exec	forall, kernel (For), scan	Parallel execution within *existing parallel region*; i.e., apply 'omp for' pragma.
omp_for_static_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp for schedule(static, ChunkSize)'
omp_for_nowait_static_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp for schedule(static, ChunkSize) nowait'
omp_for_dynamic_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp for schedule(dynamic, ChunkSize)'
omp_for_guided_exec<ChunkSize>	forall, kernel (For)	Same as applying 'omp for schedule(guided, ChunkSize)'
omp_for_runtime_exec	forall, kernel (For)	Same as applying 'omp for schedule(runtime)'

Important: RAJA only provides a nowait policy option for static schedule since that is the only schedule case that can be used with nowait and be correct in general when chaining multiple loops in a single parallel region. Paraphrasing the OpenMP standard: *programs that depend on which thread executes a particular loop iteration under any circumstance other than static schedule are non-conforming.*

Note: As in the RAJA full policies for OpenMP scheduling, the `ChunkSize` is optional. If not provided, the default chunk size that the OpenMP implementation applies will be used. For this case, the RAJA policy syntax is `omp_for_{static|dynamic|guided}_exec< >`, which will result in the OpenMP pragma `omp for schedule({static|dynamic|guided})` being applied. Similarly, for `nowait` static policy, the RAJA policy syntax is `omp_for_nowait_static_exec< >`, which will result in the OpenMP pragma `omp for schedule(static) nowait` being applied.

Note: As noted above, RAJA inner OpenMP policies must only be used within an **existing** parallel region to work properly. Embedding an inner policy inside the RAJA outer `omp_parallel_exec` will allow you to apply the OpenMP execution prescription specified by the policies to a single kernel. To support use cases with multiple kernels inside an OpenMP parallel region, RAJA provides a **region** construct that takes a template argument to specify the execution back-end. For example:

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

  RAJA::forall<RAJA::omp_for_nowait_static_exec< > >(segment,
    [=] (int idx) {
      // do something at iterate 'idx'
    }
  );

  RAJA::forall<RAJA::omp_for_static_exec< > >(segment,
    [=] (int idx) {
      // do something else at iterate 'idx'
    }
  );

});
```

Here, the `RAJA::region<RAJA::omp_parallel_region>` method call creates an OpenMP parallel region, which contains two `RAJA::forall` kernels. The first uses the `RAJA::omp_for_nowait_static_exec< >` policy, meaning that no thread synchronization is needed after the kernel. Thus, threads can start working on the second kernel while others are still working on the first kernel. I general, this will be correct when the segments used in the two kernels are the same, each loop is data parallel, and static scheduling is applied to both loops. The second kernel uses the `RAJA::omp_for_static_exec` policy, which means that all threads will complete before the kernel exits. In this example, this is not really needed since there is no more code to execute in the parallel region and there is an implicit barrier at the end of it.

Threading Building Block (TBB) Parallel CPU Policies

RAJA provides a basic set of TBB execution policies for users who would like to try it.

Threading Building Blocks Policies	Works with	Brief description
tbb_for_exec	forall, kernel (For), scan	Execute loop iterations. as tasks in parallel using TBB `parallel_for` method.
tbb_for_static<CHUNK_SIZE>	forall, kernel (For), scan	Same as above, but use. a static scheduler with given chunk size.
tbb_for_dynamic	forall, kernel (For), scan	Same as above, but use 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.

GPU Policies for CUDA and HIP

RAJA policies for GPU execution using CUDA or HIP are essentially identical. The only difference is that CUDA policies have the prefix `cuda_` and HIP policies have the prefix `hip_`.

CUDA/HIP Execution Policies	Works with	Brief description
cuda/hip_exec<BLOCK_SIZE>	forall, scan, sort	Execute loop iterations in a GPU kernel launched with given thread-block size. Note that the thread-block size must be provided, there is no default provided.
cuda/hip_thread_x_direct	kernel (For)	Map loop iterates directly to GPU threads in x-dimension, one iterate per thread (see note below about limitations)
cuda/hip_thread_y_direct	kernel (For)	Same as above, but map to threads in y-dim
cuda/hip_thread_z_direct	kernel (For)	Same as above, but map to threads in z-dim
cuda/hip_thread_x_loop	kernel (For)	Similar to thread-x-direct policy, but use a block-stride loop which doesn't limit number of loop iterates
cuda/hip_thread_y_loop	kernel (For)	Same as above, but for threads in y-dimension
cuda/hip_thread_z_loop	kernel (For)	Same as above, but for threads in z-dimension
cuda/hip_flatten_block_threads_{xyz}	Teams (Loop)	Reshapes threads in a multi-dimensional thread team into one-dimension, accepts any permutation of dimensions (expt namespace)
cuda/hip_block_x_direct	kernel (For)	Map loop iterates directly to GPU thread blocks in x-dimension, one iterate per block
cuda/hip_block_y_direct	kernel (For)	Same as above, but map to blocks in y-dimension
cuda/hip_block_z_direct	kernel (For)	Same as above, but map to blocks in z-dimension
cuda/hip_block_x_loop	kernel (For)	Similar to block-x-direct policy, but use a grid-stride loop.
cuda/hip_block_y_loop	kernel (For)	Same as above, but use blocks in y-dimension
cuda/hip_block_z_loop	kernel (For)	Same as above, but use blocks in z-dimension
cuda/hip_global_thread_x	Teams (Loop)	Creates a unique thread id for each thread on the x dimension of the grid (expt namespace)
cuda/hip_global_thread_y	Teams (Loop)	Same as above, but uses threads in y-dimension (expt namespace)
cuda/hip_global_thread_z	Teams (Loop)	Same as above, but uses threads in z-dimension (expt namespace)
cuda/hip_warp_direct	kernel (For)	Map work to threads in a warp directly. Cannot be used in conjunction with cuda/hip_thread_x_* policies. Multiple warps can be created by using cuda/hip_thread_y/z_* policies.
cuda/hip_warp_loop	kernel (For)	Policy to map work to threads in a warp using a warp-stride loop. Cannot be used in conjunction with cuda/hip_thread_x_* policies. Multiple warps can be created by using cuda/hip_thread_y/z_* policies.
cuda/hip_warp_masked_direct<BitMask<..>>	kernel (For)	Policy to map work directly to threads in a warp using a bit mask. Cannot be used in conjunction with cuda/hip_thread_x_* policies. Multiple warps can be created by using cuda/hip_thread_y/z_* policies.

Several notable constraints apply to RAJA CUDA/HIP *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 cannot be greater than the maximum allowable threads per block. Typically, this is equ:math:*leq* 1024; e.g., attempting to launch a CUDA kernel with more than 1024 threads per block will cause the CUDA runtime to complain about *illegal launch parameters.*
- **Thread-direct policies are recommended only for certain loop patterns, such as tiling.**

Several notes regarding CUDA/HIP 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/HIP thread and block loop policies are recommended for most loop patterns.**

Finally

Note: CUDA/HIP block-direct policies may be preferable to block-loop policies in situations where block load balancing may be an issue as the block-direct policies may yield better performance.

GPU Policies for SYCL

SYCL Execution Policies	Works with	Brief description
sycl_exec<WORK_GROUP_SIZE>	forall,	Execute loop iterations in a GPU kernel launched with given work group size.
sycl_global_0<WORK_GROUP_SIZE>	kernel (For)	Map loop iterates directly to GPU global ids in first dimension, one iterate per work item. Group execution into work groups of given size.
sycl_global_1<WORK_GROUP_SIZE>	kernel (For)	Same as above, but map to global ids in second dim
sycl_global_2<WORK_GROUP_SIZE>	kernel (For)	Same as above, but map to global ids in third dim
sycl_local_0_direct	kernel (For)	Map loop iterates directly to GPU work items in first dimension, one iterate per work item (see note below about limitations)
sycl_local_1_direct	kernel (For)	Same as above, but map to work items in second dim
sycl_local_2_direct	kernel (For)	Same as above, but map to work items in third dim
sycl_local_0_loop	kernel (For)	Similar to local-1-direct policy, but use a work group-stride loop which doesn't limit number of loop iterates
sycl_local_1_loop	kernel (For)	Same as above, but for work items in second dimension
sycl_local_2_loop	kernel (For)	Same as above, but for work items in third dimension
sycl_group_0_direct	kernel (For)	Map loop iterates directly to GPU group ids in first dimension, one iterate per group
sycl_group_1_direct	kernel (For)	Same as above, but map to groups in second dimension
sycl_group_2_direct	kernel (For)	Same as above, but map to groups in third dimension
sycl_group_0_loop	kernel (For)	Similar to group-1-direct policy, but use a group-stride loop.
sycl_group_1_loop	kernel (For)	Same as above, but use groups in second dimension
sycl_group_2_loop	kernel (For)	Same as above, but use groups in third dimension

OpenMP Target Offload Policies

RAJA provides policies to use OpenMP to offload kernel execution to a GPU device, for example. They are summarized in the following table.

OpenMP Target Execution Policies	Works with	Brief description
omp_target_parallel_for_exec<#>	forall, kernel(For)	Create parallel target region and execute with given number of threads per team inside it. Number of teams is calculated internally; i.e., apply `omp teams distribute parallel for num_teams(iteration space size/#) thread_limit(#)` pragma
omp_target_parallel_collapse_exec	kernel (Collapse)	Similar to above, but collapse *perfectly-nested* loops, indicated in arguments to RAJA Collapse statement. Note: compiler determines number of thread teams and threads per team

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 outer 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

Earlier, we discussed an example using the `RAJA::region` construct to execute multiple kernels in an OpenMP parallel region. To support source code portability, RAJA provides a sequential region concept that can be used to surround code that uses execution back-ends other than OpenMP. For example:

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

   RAJA::forall<RAJA::loop_exec>(segment, [=] (int idx) {
       // do something at iterate 'idx'
   } );

   RAJA::forall<RAJA::loop_exec>(segment, [=] (int idx) {
       // do something else at iterate 'idx'
   } );

 });
```

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, for example, 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/hip_reduce	any CUDA/HIP policy	Parallel reduction in a CUDA/HIP kernel (device synchronization will occur when reduction value is finalized).
cuda/hip_reduce_atomic	any CUDA/HIP policy	Same as above, but reduction may use CUDA atomic operations.
sycl_reduce	any SYCL policy	Reduction in a SYCL kernel (device synchronization will occur when the reduction value is finalized).

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/hip_atomic	any CUDA/HIP policy	Atomic operation performed in a CUDA/HIP kernel.
cuda/hip_atomic_explicit	any CUDA/HIP policy	Atomic operation performed in a CUDA/HIP kernel that may also be used in a host execution context. The atomic policy takes a host atomic policy template argument. See additional explanation and example below.
builtin_atomic	seq_exec, loop_exec, any OpenMP policy	Compiler *builtin* atomic operation.
auto_atomic	seq_exec, loop_exec, any OpenMP policy, any CUDA/HIP policy	Atomic operation *compatible* with loop execution policy. See example below. Can not be used inside cuda/hip explicit atomic policies.

Note: The `cuda_atomic_explicit` and `hip_atomic_explicit` policies take a host atomic policy template parameter. They are intended to be used with kernels that are host-device decorated to be used in either a host or device execution context.

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

```
auto kernel = [=] RAJA_HOST_DEVICE (RAJA::Index_type i) {
  RAJA::atomicAdd< RAJA::cuda_atomic_explicit<omp_atomic> >(&sum, 1);
};

RAJA::forall< RAJA::cuda_exec<BLOCK_SIZE> >(RAJA::RangeSegment seg(0, N), kernel);

RAJA::forall< RAJA::omp_parallel_for_exec >(RAJA::RangeSegment seg(0, N),
    kernel);
```

In this case, the atomic operation knows when it is compiled for the device in a CUDA kernel context and the CUDA atomic operation is applied. Similarly when it is compiled for the host in an OpenMP kernel the omp_atomic policy is used and the OpenMP version of the atomic operation is applied.

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

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

  RAJA::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/hip_shared_mem` - Allocate CUDA or HIP shared memory
- `RAJA::cuda/hip_thread_mem` - Allocate CUDA or HIP 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++14 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>
    >
  >;
```

(continues on next page)

(continued from previous page)

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

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:

- RAJA::kernel_param functions similar to RAJA::kernel

except that the second argument is a *tuple of parameters* used in a kernel for local arrays, thread local variables, tiling information, etc.

Note:

- All of the statement types described below are in the namespace

RAJA::statement. For breavity, we omit the namespaces.

Several RAJA statements can be specialized with auxilliary types, which are described in *Auxilliary Types*.

The following list contains the most commonly used statement types.

- 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.
- Lambda< LambdaId > invokes the lambda expression that appears at position 'LambdaId' in the sequence of lambda arguments. With this statement, the lambda expression must accept all arguments associated with the tuple of iteration space segments and tuple of parameters (if kernel_param is used).
- Lambda< LambdaId, Args...> extends the Lambda statement. The second template parameter indicates which arguments (e.g., which segment iteration variables) are passed to the lambda expression.
- 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.

There is one statement specific to OpenMP kernels.

- `OmpSyncThreads` applies the OpenMP `#pragma omp barrier` directive.

Statement types that lauch CUDA or HIP GPU kernels are listed next. They work similarly for each back-end and their names are distinguished by the prefix `Cuda` or `Hip`. For example, `CudaKernel` or `HipKernel`.

- `Cuda/HipKernel< EnclosedStatements>` launches ‘‘EnclosedStatements’ as a GPU 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. This kernel launch is synchronous.
- `Cuda/HipKernelAsync< EnclosedStatements>` asynchronous version of Cuda/HipKernel.
- `Cuda/HipKernelFixed<num_threads, EnclosedStatements>` similar to Cuda/HipKernel but enables a fixed number of threads (specified by num_threads). This kernel launch is synchronous.
- `Cuda/HipKernelFixedAsync<num_threads, EnclosedStatements>` asynchronous version of Cuda/HipKernelFixed.
- `CudaKernelFixedSM<num_threads, min_blocks_per_sm, EnclosedStatements>` similar to CudaKernelFixed but enables a minimum number of blocks per sm (specified by min_blocks_per_sm), this can help increase occupancy. This kernel launch is synchronous. **Note: there is no HIP variant of this statement.**
- `CudaKernelFixedSMAsync<num_threads, min_blocks_per_sm, EnclosedStatements>` asynchronous version of CudaKernelFixedSM. **Note: there is no HIP variant of this statement.**
- `Cuda/HipKernelOcc<EnclosedStatements>` similar to CudaKernel but uses the CUDA occupancy calculator to determine the optimal number of threads/blocks. Statement is intended for use with RAJA::cuda/hip_block_{xyz}_loop policies. This kernel launch is synchronous.
- `Cuda/HipKernelOccAsync<EnclosedStatements>` asynchronous version of Cuda/HipKernelOcc.
- `Cuda/HipKernelExp<num_blocks, num_threads, EnclosedStatements>` similar to CudaKernelOcc but with the flexibility to fix the number of threads and/or blocks and let the CUDA occupancy calculator determine the unspecified values. This kernel launch is synchronous.
- `Cuda/HipKernelExpAsync<num_blocks, num_threads, EnclosedStatements>` asynchronous version of Cuda/HipKernelExp.
- `Cuda/HipSyncThreads` invokes CUDA or HIP ‘__syncthreads()’ barrier.
- `Cuda/HipSyncWarp` invokes CUDA ‘__syncwarp()’ barrier. **Note: warp sync is not supported, so the HIP variant is a no-op.

Statement types that lauch SYCL kernels are listed next.

- `SyclKernel<EnclosedStatements>` launches `EnclosedStatements` as a SYCL kernel. This kernel launch is synchronous.
- `SyclKernelAsync<EnclosedStatements>` asynchronous version of SyclKernel.

RAJA provides statements to define loop tiling which can improve performance; e.g., by allowing CPU cache blocking or use of GPU shared memory.

- `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.
- `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.

- `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.

It is often advantageous to use local arrays for data accessed in tiled loops. RAJA provides a statement for allocating data in a *Local Array* object according to a memory policy. See *Local Array Memory Policies* for more information about such policies.

- `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.

RAJA provides some statement types that apply in specific kernel scenarios.

- `Reduce< ReducePolicy, Operator, ParamId, EnclosedStatements >` reduces a value across threads in a multi-threaded code region 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.
- `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.
- `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.

Auxilliary Types

The following list summarizes auxillary types used in the above statments. These types live in the `RAJA` namespace.

- `tile_fixed<TileSize>` tile policy argument to a `Tile` or `TileTCount` statement; 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.
- `tile_dynamic<ParamIdx>` TilePolicy argument to a Tile or TileTCount statement; partitions loop iterations into tiles of a size specified by a `TileSize{}` positional parameter argument. This statement type can be used as the `TilePolicy` template paramter in the `Tile` statements above.
- `Segs<...>` argument to a Lambda statement; used to specify which segments in a tuple will be used as lambda arguments.
- `Offsets<...>` argument to a Lambda statement; used to specify which segment offsets in a tuple will be used as lambda arguments.

- `Params<...>` argument to a Lambda statement; used to specify which params in a tuple will be used as lambda arguments.
- `ValuesT<T, ...>` argument to a Lambda statement; used to specify compile time constants, of type T, that will be used as lambda arguments.

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

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 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 the `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.

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.

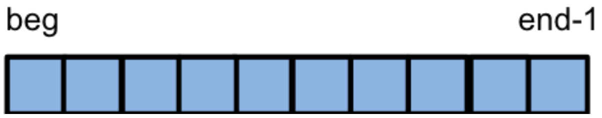

Fig. 1: A range segment defines a stride-1 index range [beg, end).

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 similar to a C-style for-loop.

Strided Segments

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

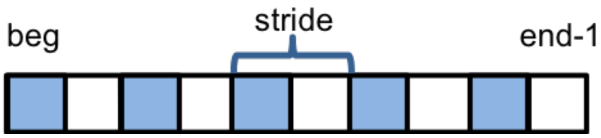

Fig. 2: A range-stride segment defines an index range with arbitrary stride [beg, end, stride).

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.

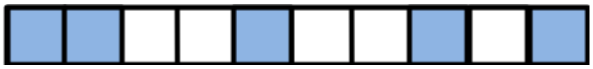

Fig. 3: A list segment defines an arbitrary collection of indices. Here, we have a list segment with 5 irregularly-spaced indices.

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 where the indices are
// stored in the host memory space
camp::resources::Resource host_res{camp::resources::Host()};
RAJA::TypedListSegment<int> idx_list( &idx[0], idx.size(),
                                      host_res );
```

Using a list segment in a RAJA loop traversal template will run the loop indices specified in the array passed to the list segment constructor. That is, using 'idx_list' from above:

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

will print the values:

```
0 2 3 4 7 8 9 53
```

Note that a `RAJA::TypedListSegment` constructor can take a pointer to an array of indices and an array length, as shown above. If the indices are in a container, such as `std::vector` that provides `begin()`, `end()`, and `size()` methods, the length argument is not required. For example:

```
std::vector<int> idx = {0, 2, 3, 4, 7, 8, 9, 53};

camp::resources::Resource host_res{camp::resources::Host()};
RAJA::TypedListSegment<int> idx_list( idx, host_res );
```

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.

By default, the list segment constructor copies the indices in the array passed to it to the memory space specified by the resource argument. The resource argument is required so that the segment index values are in the proper memory space for the kernel to run. Since the kernel is run on the CPU host in this example (indicated by the `RAJA::seq_exec` execution policy), we pass a host resource object to the list segment constructor. If, for example, the kernel was to run on a GPU using a CUDA or HIP execution policy, then the resource type passed to the camp resource constructor would be `camp::resources::Cuda()` or `camp::resources::Hip()`, respectively.

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.

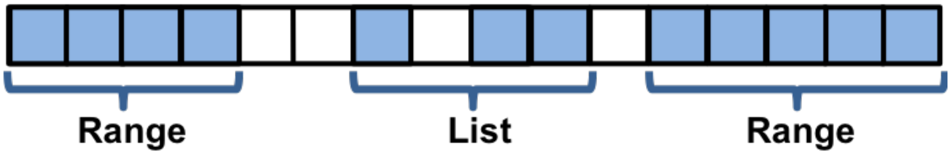

Fig. 4: An index set with 2 range segments and one list segment.

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) { ... });
```

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: Iterating over the indices of all segments in a RAJA index set requires a two-level execution policy, with two template parameters, as shown above. The first parameter specifies how to iterate over the seqments. The second parameter specifies how each segment will execute. See *RAJA IndexSet Execution Policies* for more information about RAJA index set execution policies.

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.

View and Layout

Matrices and tensors, which are common in scientific computing applications, are naturally expressed 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 to access the corresponding array memory location. One could use 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 may be 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 Views

A `RAJA::View` object wraps a pointer and enables indexing into the data referenced via the pointer based on 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 follows the C and C++ standards for multi-dimensional arrays) through the view *parenthesis operator*:

```
// r - row index of matrix
// c - column index of 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 $N_n * N_{(n-1)} * \ldots * N_{(j+1)}$.

MultiView

Using numerous arrays with the same size and Layout, where each needs a View, can be cumbersome. Developers need to create a View object for each array, and when using the Views in a kernel, they require redundant pointer offset calculations. `RAJA::MultiView` solves these problems by providing a way to create many Views with the same Layout in one instantiation, and operate on an array-of-pointers that can be used to succinctly access data.

A `RAJA::MultiView` object wraps an array-of-pointers, or a pointer-to-pointers, whereas a `RAJA::View` wraps a single pointer or array. This allows a single `RAJA::Layout` to be applied to multiple arrays associated with the MultiView, allowing the arrays to share indexing arithmetic when their access patterns are the same.

The instantiation of a MultiView works exactly like a standard View, except that it takes an array-of-pointers. In the following example, a MultiView applies a 1-D layout of length 4 to 2 arrays in `myarr`.

```
  // Arrays of the same size, which will become internal to the MultiView.
  int a1[4] = {5,6,7,8};
  int a2[4] = {9,10,11,12};

  // Array-of-pointers which will be passed into MultiView.
  int * myarr[2];
  myarr[0] = a1;
  myarr[1] = a2;

  // This MultiView applies a 1-D layout of length 4 to each internal array in myarr.
  RAJA::MultiView< int, RAJA::Layout<1> > MView(myarr, 4);
```

The default MultiView accesses individual arrays via the 0-th position of the MultiView.

```
  t1 = MView( 0, 3 ); // accesses the 4th index of the 0th internal array a1, returns␣
↪value of 8
  t2 = MView( 1, 2 ); // accesses 3rd index of the 1st internal array a2, returns␣
↪value of 11
```

The index into the array-of-pointers can be moved to different argument positions of the MultiView `()` access operator, rather than the default 0-th position. For example, by passing a third template argument to the MultiView constructor in the previous example, the internal array index and the integer indicating which array to access can be reversed.

```
  // MultiView with array-of-pointers index in 1st position.
  RAJA::MultiView< int, RAJA::Layout<1>, 1 > MView1(myarr, 4);
```

(continues on next page)

(continued from previous page)

```
  t3 = MView1( 3, 0 ); // accesses the 4th index of the 0th internal array a1,␣
↪returns value of 8
  t4 = MView1( 2, 1 ); // accesses 3rd index of the 1st internal array a2, returns␣
↪value of 11
```

With higher dimensional Layouts, the index into the array-of-pointers can be moved to other positions in the MultiView `()` access operator. Here is an example that compares the accesses of a 2-D layout on a normal `RAJA::View` with a `RAJA::MultiView` with the array-of-pointers index set to the 2nd position.

```
  RAJA::View< int, RAJA::Layout<2> > normalView(a1, 2, 2);

  t1 = normalView( 1, 1 ); // accesses 4th index of the a1 array, value = 8

  // MultiView with array-of-pointers index in 2nd position
  RAJA::MultiView< int, RAJA::Layout<2>, 2 > MView2(myarr, 2, 2);

  t2 = MView2( 1, 1, 0 ); // accesses the 4th index of the 0th internal array a1,␣
↪returns value of 8 (same as normalView(1,1))
  t3 = MView2( 0, 0, 1 ); // accesses the 1st index of the 1st internal array a2,␣
↪returns value of 9
```

RAJA Layouts

`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 them here.

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 properly initialize the internal sub-object which holds the extents.** The second argument is the striding permutation and similarly requires double braces.

In the next example, we create the same permuted layout as above, 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 );

// The Layout template parameters are dimension, 'linear index' type used
// when converting an index triple into the corresponding pointer offset
// index, and the index with unit stride
RAJA::View<double, RAJA::Layout<3, int, 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. **The layout permutation and unit-stride index specification must be consistent to prevent incorrect indexing.**

Offset Layout

The `RAJA::make_offset_layout` method creates a `RAJA::OffsetLayout` 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::OffsetLayout<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 an array offset index by subtracting the lower offset from it; i.e., in the loop, each 'i' value has '-5' subtracted from it to properly access the array entry. That is, the sequence of indices generated by the for-loop:

```
-5 -4 -3 ... 5
```

will index into the data array as:

```
0 1 2 ... 10
```

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::OffsetLayout<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 noted earlier, double braces are needed to properly initialize the internal data in the layout object.

Permuted Offset Layout

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

```
std::array< RAJA::idx_t, 2> perm {{1, 0}};
RAJA::OffsetLayout<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, which is the extent of the first index ($[-1, 2]$).

Note: It is important to note some facts about RAJA layout types. All layouts have a permutation. So a permuted layout and a "non-permuted" layout (i.e., default permutation) has the type `RAJA::Layout`. Any layout with an offset has the type `RAJA::OffsetLayout`. The `RAJA::OffsetLayout` type has a `RAJA::Layout` and offset data. This was an intentional design choice to avoid the overhead of offset computations in the `RAJA::View` data access operator when they are not needed.

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.

Typed Layouts

RAJA provides typed variants of `RAJA::Layout` and `RAJA::OffsetLayout` that enable users to specify integral index types. Usage requires specifying types for the linear index and the multi-dimensional indicies. The following example creates two two-dimensional typed layouts where the linear index is of type TIL and the '(x, y)' indices for accesingg the data have types TIX and TIY:

```
RAJA_INDEX_VALUE(TIX, "TIX");
RAJA_INDEX_VALUE(TIY, "TIY");
RAJA_INDEX_VALUE(TIL, "TIL");

RAJA::TypedLayout<TIL, RAJA::tuple<TIX,TIY>> layout(10, 10);
RAJA::TypedOffsetLayout<TIL, RAJA::tuple<TIX,TIY>> offLayout(10, 10);;
```

Note: Using the `RAJA_INDEX_VALUE` macro to create typed indices is helpful to prevent incorrect usage by detecting at compile when, for example, indices are passes to a view parenthesis operator in the wrong order.

Shifting Views

RAJA views include a shift method enabling users to generate a new view with offsets to the base view layout. The base view may be templated with either a standard layout or offset layout and their typed variants. The new view will use an offset layout or typed offset layout depending on whether the base view employed a typed layout. The example below illustrates shifting view indices by N,

```
int N_r = 10;
int N_c = 15;
int *a_ptr = new int[N_r * N_c];

RAJA::View<int, RAJA::Layout<DIM>> A(a_ptr, N_r, N_c);
RAJA::View<int, RAJA::OffsetLayout<DIM>> Ashift = A.shift( {{N,N}} );

for(int y = N; y < N_c + N; ++y) {
  for(int x = N; x < N_r + N; ++x) {
    Ashift(x,y) = ...
  }
}
```

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 the rightmost index 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 layouts also support *projections*, where one or more dimension extent is zero. In this case, the linear index space is invariant for those 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 zero for j
int i,j,k;
layout.toIndices(lin2, i, j, k); // i,j,k = {0, 0, 1}
```

RAJA Atomic Views

Any `RAJA::View` object can be made *atomic* so that any update to a data entry accessed via the view can only be performed one thread (CPU or GPU) at a time. For example, suppose you have an integer array of length N, whose element values are in the set {0, 1, 2, . . . , M-1}, where M < N. You want to build a histogram array of length M such that the i-th entry in the array is the number of occurrences of the value i in the original array. Here is one way to do this in parallel using OpenMP and a RAJA atomic view:

```
using EXEC_POL = RAJA::omp_parallel_for_exec;
using ATOMIC_POL = RAJA::omp_atomic

int* array = new double[N];
int* hist_dat = new double[M];

// initialize array entries to values in {0, 1, 2, ..., M-1}...
// initialize hist_dat to all zeros...

// Create a 1-dimensional view for histogram array
RAJA::View<int, RAJA::Layout<1> > hist_view(hist_dat, M);

// Create an atomic view into the histogram array using the view above
auto hist_atomic_view = RAJA::make_atomic_view<ATOMIC_POL>(hist_view);

RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N), [=] (int i) {
  hist_atomic_view( array[i] ) += 1;
} );
```

Here, we create a one-dimensional view for the histogram data array. Then, we create an atomic view from that, which we use in the RAJA loop to compute the histogram entries. Since the view is atomic, only one OpenMP thread can write to each array entry at a time.

RAJA View/Layouts Bounds Checking

The RAJA CMake variable `RAJA_ENABLE_BOUNDS_CHECK` may be used to turn on/off runtime bounds checking for RAJA views. This may be a useful debugging aid for users. When attempting to use an index value that is out of bounds, RAJA will abort the program and print the index that is out of bounds and the value of the index and bounds for it. Since the bounds checking is a runtime operation, it incurs non-negligible overhead. When bounds checkoing is turned off (default case), there is no additional run time overhead incurred.

Reduction Operations

RAJA does not provide separate loop execution methods for loops containing reduction operations like some other C++ loop programming abstraction models. Instead, RAJA provides reduction types that allow users to perform reduction operations in `RAJA::forall` and `RAJA::kernel` kernels 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. The **reduction policy type must be compatibe with the execution policy used by the kernel.** For example, in a CUDA kernel, a CUDA reduction policy must be used.
- Each RAJA reduction type accepts an **initial reduction value or values** at construction (see below).
- Each RAJA reduction type has a 'get' method to access reduced values 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.

and two less common bitwise reduction types:

- `ReduceBitAnd< reduce_policy, data_type >` - Bitwise 'and' of values (i.e., `a & b`).
- `ReduceBitOr< reduce_policy, data_type >` - Bitwise 'or' of values (i.e., `a | b`).

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 these reductions are used in a parallel execution context, the loop index computed for the reduction value may be any index where the min or max occurs.

Note: `RAJA::ReduceBitAnd` and `RAJA::ReduceBitOr` reduction types are designed to work on integral data types because **in C++, at the language level, there is no such thing as a bitwise operator on floating-point numbers.**

Reduction Examples

Next, we provide a few examples to illustrate basic usage of RAJA reduction types.

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 in the array to make the example mildly interesting...
//
```

(continues on next page)

(continued from previous page)

```
int vec[N] = {1};
vec[100] = -10; vec[500] = -10;

// Create a sum reduction object with initial value of zero
RAJA::ReduceSum< RAJA::omp_reduce, int > vsum(0);

// Create a min-loc reduction object with initial min value of 100
// and initial location index value of -1
RAJA::ReduceMinLoc< RAJA::omp_reduce, int > vminloc(100, -1);

// Run a kernel using the reduction objects
RAJA::forall<RAJA::omp_parallel_for_exec>( RAJA::RangeSegment(0, N),
  [=](RAJA::Index_type i) {

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

});

// After kernel is run, extract the reduced values
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. Also, note that the reduction objects are created using a `RAJA::omp_reduce` reduction policy, which is compatible with the OpenMP execution policy used in the kernel.

Here is an example of a bitwise or reduction:

```
const int N = 100;

//
// Initialize all entries in array of length N to the value '9'
//
int vec[N] = {9};

// Create a bitwise or reduction object with initial value of '5'
RAJA::ReduceBitOr< RAJA::omp_reduce, int > my_or(5);

// Run a kernel using the reduction object
RAJA::forall<RAJA::omp_parallel_for_exec>( RAJA::RangeSegment(0, N),
  [=](RAJA::Index_type i) {

  my_or |= vec[i];

});
```

(continues on next page)

(continued from previous page)

```
// After kernel is run, extract the reduced value
int my_or_reduce_val = static_cast<int>(my_or.get());
```

The result of the reduction is the value '13'. In binary representation (i.e., bits), $9 = ...01001$ (the vector entries) and $5 = ...00101$ (the initial reduction value). So $9|5 = ...01001|...00101 = ...01101 = 13$.

Reduction Policies

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

Resources

This section describes the basic concepts of Resource types and their functionality in `RAJA::forall`. Resources are used as an interface to various backend constructs and their respective hardware. Currently there exists Resource types for `Cuda`, `Hip`, `Omp` (target) and `Host`. Resource objects allow the user to execute `RAJA::forall` calls asynchronously on a respective thread/stream. The underlying concept of each individual Resource is still under development and it should be considered that functionality / behaviour may change.

Note:

- Currently feature complete asynchronous behavior and streamed/threaded support is available only for `Cuda` and `Hip` resources.
- The `RAJA::resources` namespace aliases the `camp::resources` namespace.

Each resource has a set of underlying functionality that is synonymous across all resource types.

Methods	Brief description
get_platform	Returns the underlying camp platform the resource is associated with.
get_event	Return an Event object for the resource from the last resource call.
allocate	Allocate data per the resource's given backend.
deallocate	Deallocate data per the resource's given backend.
memcpy	Perform a memory copy from a src location to a destination location from the resource's backend.
memset	Set memory value per the resourse's given backend.
wait_for	Enqueue a wait on the resource's stream/thread for a user passed event to occur.

Note: `deallocate`, `memcpy` and `memset` will only work with pointers that correspond to memory locations that have been allocated on the resource's respective device.

Each resource type also defines specific backend information/functionality. For example, each CUDA resource contains a `cudaStream_t` value with an associated get method. See the individual functionality for each resource in `raja/tpl/camp/include/resource/`.

Note: Stream IDs are assigned to resources in a round robin fashion. The number of independent streams for a given backend is limited to the maximum number of concurrent streams that the back-end supports.

Type-Erasure

Resources can be declared in two formats: An erased resource, and a concrete resource. The underlying runtime functionality is the same for both formats. An erased resource allows a user the ability to change the resource backend at runtime.

Concrete CUDA resource:

```
RAJA::resources::Cuda my_cuda_res;
```

Erased resource:

```
if (use_gpu)
  RAJA::resources::Resource my_res{RAJA::resources::Cuda()};
else
  RAJA::resources::Resource my_res{RAJA::resources::Host()};
```

Memory allocation on resources:

```
int* a1 = my_cuda_res.allocate<int>(ARRAY_SIZE);
int* a2 = my_res.allocate<int>(ARRAY_SIZE);
```

If `use_gpu` is `true`, then the underlying type of `my_res` is a CUDA resource. Therefore `a1` and `a2` will both be allocated on the GPU. If `use_gpu` is `false`, then only `a1` is allocated on the GPU, and `a2` is allocated on the host.

Forall

A resource is an optional argument to a `RAJA::forall` call. When used, it is passed as the first argument to the method:

```
RAJA::forall<ExecPol>(my_gpu_res, .... )
```

When specifying a CUDA or HIP resource, the `RAJA::forall` is executed aynchronously on a stream. Currently, CUDA and HIP are the only Resources that enable asynchronous threading with a `RAJA::forall`. All other calls default to using the `Host` resource until further support is added.

The Resource type that is passed to a `RAJA::forall` call must be a concrete type. This is to allow for a compile-time assertion that the resource is not compatible with the given execution policy. For example:

```
using ExecPol = RAJA::cuda_exec_async<BLOCK_SIZE>;
RAJA::resources::Cuda my_cuda_res;
RAJA::resources::Resource my_res{RAJA::resources::Cuda()};
RAJA::resources::Host my_host_res;

RAJA::forall<ExecPol>(my_cuda_res, .... ) // Compiles.
RAJA::forall<ExecPol>(my_res, .... )      // Compilation Error. Not Concrete.
RAJA::forall<ExecPol>(my_host_res, .... ) // Compilation Error. Mismatched Resource␣
↪and Exec Policy.
```

Below is a list of the currently available concrete resource types and their execution policy suport.

Resource	Policies supported
Cuda	cuda_exec cuda_exec_async cuda_exec_explicit
Hip	hip_exec hip_exec_async
Omp*	omp_target_parallel_for_exec omp_target_parallel_for_exec_n
Host	loop_exec seq_exec openmp_parallel_exec omp_for_schedule_exec omp_for_nowait_schedule_exec simd_exec tbb_for_dynamic tbb_for_static

Note: The `RAJA::resources::Omp` resource is still under development.

IndexSet policies require two execution policies (see *IndexSets*). Currently, users may only pass a single resource to a forall method taking an IndexSet argument. This resource is used for the inner execution of each Segment in the IndexSet:

```
using ExecPol = RAJA::ExecPolicy<RAJA::seq_segit, RAJA::cuda_exec<256>>;
RAJA::forall<ExecPol>(my_cuda_res, iset,  .... );
```

When a resource is not provided by the user, a *default* resource is assigned, which can be accessed in a number of ways. It can be accessed directly from the concrete resource type:

```
RAJA::resources::Cuda my_default_cuda = RAJA::resources::Cuda::get_default();
```

The resource type can also be deduced from an execution policy:

```
using Res = RAJA::resources::get_resource<ExecPol>::type;
Res r = Res::get_default();
```

Finally, the resource type can be deduced from an execution policy:

```
auto my_resource = RAJA::resources::get_default_resource<ExecPol>();
```

Note: For CUDA and HIP, the default resource is *NOT* the CUDA or HIP default stream. It is its own stream defined in `camp/include/resource/`. This is an attempt to break away from some of the issues that arise from the synchronization behaviour of the CUDA and HIP default streams. It is still possible to use the CUDA and HIP default streams as the default resource. This can be enabled by defining the environment variable `CAMP_USE_PLATFORM_DEFAULT_STREAM` before compiling RAJA in a project.

Events

Event objects allow users to wait or query the status of a resource's action. An event can be returned from a resource:

```
RAJA::resources::Event e = my_res.get_event();
```

Getting an event like this enqueues an event object for the given back-end.

Users can call the *blocking* `wait` function on the event:

```
e.wait();
```

Preferably, users can enqueue the event on a specific resource, forcing only that resource to wait for the event:

```
my_res.wait_for(&e);
```

The usage allows one to set up dependencies between resource objects and `RAJA::forall` calls.

Note: An Event object is only created if a user explicitly sets the event returned by the `RAJA::forall` call to a variable. This avoids unnecessary event objects being created when not needed. For example:

```
forall<cuda_exec_async<BLOCK_SIZE>>(my_cuda_res, ...
```

will *not* generate a cudaStreamEvent, whereas:

```
RAJA::resources::Event e = forall<cuda_exec_async<BLOCK_SIZE>>(my_cuda_res, ...
```

will generate a cudaStreamEvent.

Example

This example executes three kernels across two cuda streams on the GPU with a requirement that the first and second kernel finish execution before launching the third. It also demonstrates copying memory from the device to host on a resource:

First, define two concrete CUDA resources and one host resource:

```
  RAJA::resources::Cuda res_gpu1;
  RAJA::resources::Cuda res_gpu2;
  RAJA::resources::Host res_host;

  using EXEC_POLICY = RAJA::cuda_exec_async<GPU_BLOCK_SIZE>;
```

Next, allocate data for two device arrays and one host array:

```
int* d_array1 = res_gpu1.allocate<int>(N);
int* d_array2 = res_gpu2.allocate<int>(N);
int* h_array  = res_host.allocate<int>(N);
```

Then, Execute a kernel on CUDA stream 1 `res_gpu1`:

```
RAJA::forall<EXEC_POLICY>(res_gpu1, RAJA::RangeSegment(0,N),
  [=] RAJA_HOST_DEVICE (int i) {
    d_array1[i] = i;
  }
);
```

and execute another kernel on CUDA stream 2 `res_gpu2` storing a handle to an `Event` object to a local variable:

```
RAJA::resources::Event e = RAJA::forall<EXEC_POLICY>(res_gpu2, RAJA::RangeSegment(0,
↪N),
  [=] RAJA_HOST_DEVICE (int i) {
    d_array2[i] = -1;
  }
);
```

The next kernel on `res_gpu1` requires that the last kernel on `res_gpu2` finish first. Therefore, we enqueue a wait on `res_gpu1` that enforces this:

```
res_gpu2.wait_for(&e);
```

Execute the second kernel on `res_gpu1` now that the two previous kernels have finished:

```
RAJA::forall<EXEC_POLICY>(res_gpu1, RAJA::RangeSegment(0,N),
  [=] RAJA_HOST_DEVICE (int i) {
    d_array1[i] *= d_array2[i];
  }
);
```

We can enqueue a memcpy operation on `res_gpu1` to move data from the device to the host:

```
res_gpu1.memcpy(h_array, d_array1, sizeof(int) * N);
```

Lastly, we use the copied data on the host side:

```
bool check = true;
RAJA::forall<RAJA::seq_exec>(res_host, RAJA::RangeSegment(0,N),
  [&check, h_array] (int i) {
    if(h_array[i] != -i) {check = false;}
  }
);
```

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 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.
- See *Atomics* for details about CUDA atomic operations.

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

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

});
```

After this kernel executes, the value reference by 'sum' will be '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::AtomicRef` provides an object-oriented interface to the atomic methods described above. For example, after the following operations:

```
double val = 2.0;
RAJA::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*.

CUDA Atomics Architecture Dependencies

The internal implementations for RAJA atomic operations may vary depending on which CUDA architecture is available and/or specified when the RAJA is configured for compilation. The following rules apply when the following CUDA architecture level is chosen:

- **CUDA architecture is lower than 'sm_35'**
  - Certain atomics will be implemented using CUDA *atomicCAS* (Compare and Swap).
- **CUDA architecture is 'sm_35' or higher**
  - CUDA native 64-bit unsigned atomicMin, atomicMax, atomicAnd, atomicOr, atomicXor are used.
- **CUDA architecture is 'sm_60' or higher**
  - CUDA native 64-bit double *atomicAdd* is used.

DESUL Atomics Support

RAJA provides support for the use of DESUL Atomics as an alternative backend to the default implementation of RAJA atomics. DESUL atomics are considered an **experimental** feature in RAJA at this point. DESUL atomics may impact the performance of some atomic functions. While switching to DESUL atomics typically yields positive or neutral performance results, some atomic operations may perform worse when using DESUL.

To enable DESUL Atomics:

1. Ensure that RAJA and its dependencies are configured to use C++14.
2. Set `RAJA_ENABLE_DESUL_ATOMICS=On`.

Enabling DESUL Atomics alters RAJA atomic functions to be wrapper-functions for their DESUL counterparts. This removes the need for user code changes to switch between DESUL and RAJA implementations. The exception to this is when RAJA atomic helper functions are used instead of the backwards-compatible API functions specified by *Atomic Operations*. By *helper functions*, we mean the RAJA atomic methods which take a reduction policy object as the first argument, instead of specifying the reduction policy type as a template parameter.

DESUL atomic functions are compiled with the proper back-end implementation based on the scope in which they are called, which removes the need to specify atomic policies for target back-ends. As a result, atomic policies such as `cuda_atomic` or `omp_atomic` are ignored when DESUL is enabled, but are still necessary to pass in as parameters to the RAJA API. This will likely change in the future and RAJA atomic policies will be removed.

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 NVIDIA CUB library internally. The CMake variable `CUB_DIR` will be automatically set to the location of the CUB library when CUDA is enabled. Details for using a different version of the CUB library are available in the *Getting Started With RAJA* section.

Note: For scans using the HIP back-end, RAJA uses the AMD rocPRIM library internally. The CMake variable `ROCPRIM_DIR` will be automatically set to the location of the rocPRIM library when HIP is enabled. Details for using a different version of the rocPRIM library are available in the *Getting Started With RAJA* section.

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

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

and calculates the output sequence:

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

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_container, out_container)`
- `RAJA::inclusive_scan< exec_policy >(in_container, out_container, operator)`

Here, 'in_container' and 'out_container' are random access ranges 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. Note that container arguments can be generated from iterators using `RAJA::make_span(begin, len)`.

RAJA also provides *in-place* scans:

- `RAJA::inclusive_scan_inplace< exec_policy >(in_container)`
- `RAJA::inclusive_scan_inplace< exec_policy >(in_container, <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_container, out_container)`
- `RAJA::exclusive_scan< exec_policy >(in_container, out_container, operator)`

and

- `RAJA::exclusive_scan_inplace< exec_policy >(in_container)`
- `RAJA::exclusive_scan_inplace< exec_policy >(in_container, <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*.

Sorts

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

A few important notes:

Note:

- All RAJA sort operations are in the namespace `RAJA`.

- Each RAJA sort operation is a template on an *execution policy* parameter. The same policy types used for `RAJA::forall` methods may be used for RAJA sorts.
- RAJA sort operations accept an optional *comparator* argument so users can perform different types of sort operations. If no operator is given, the default is a *less than* operation and the result is **non-decreasing**.

Also:

Note:

- For sorts using the CUDA back-end, RAJA uses the implementations provided by the NVIDIA CUB library. For information please see *build-external-tpl*.
- For sorts using the HIP back-end, RAJA uses the implementations provided by the AMD rocPRIM library. For information please see *build-external-tpl*.
- The RAJA CUDA and HIP back-ends only support sorting arithmetic types using RAJA operators 'less than' and 'greater than'.

Please see the *Parallel Sort Operations* tutorial section for usage examples of RAJA sort operations.

Sort Operations

In general, a sort operation takes a sequence of numbers `x` and a binary comparison operator `op` that forms a strict weak ordering of elements in input sequence `x` and produces a sequence of numbers `y` as output. The output sequence is a permutation of the input sequence where each pair of elements `a` and `b`, where `a` is before `b` in the output sequence, satisfies `!(b op a)`. Sorts are stable if they always preserve the order of equivalent elements, where equivalent elements satisfy `!(a op b) && !(b op a)`.

A **stable sort** takes an input sequence `x` where a_i appears before a_j if i < j when a_i and a_j are equivalent for any i != j.

x = { a_0, b_0, a_1, ... }

and calculates the stably sorted output sequence `y` that preserves the order of equivalent elements. That is, the sorted sequence where element a_i appears before the equivalent element a_j if i < j:

y = { a_0, a_1, b_0, ... }

An **unstable sort** may not preserve the order of equivalent elements and may produce either of the following output sequences:

y = { a_0, a_1, b_0, ... }

or

y = { a_1, a_0, b_0, ... }

RAJA Unstable Sorts

RAJA unstable sort operations look like the following:

- `RAJA::sort< exec_policy >(container)`
- `RAJA::sort< exec_policy >(container, comparator)`

For example, sorting an array with this sequence of values:

```
6 7 2 1 0 9 4 8 5 3 4 9 6 3 7 0 1 8 2 5
```

with a sequential unstable sort operation:

```
  RAJA::sort<RAJA::seq_exec>(RAJA::make_span(out, N));
```

produces the `out` array with this sequence of values:

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

Note that the syntax is essentially the same as *Parallel Scan Operations*. Here, `container` is a random access range of elements. `container` provides access to the input sequence and contains the output sequence at the end of sort. The first sort operation listed above will be a *non-decreasing* sort since there is no comparator argument given; i.e., the sequences will be reordered *in-place* using operator::less. The second sort will apply the comparator that is passed into the function. Note that the container argument can be generated from iterators using `RAJA::make_span(begin, len)`.

RAJA also provides sort operations that operate on key-value pairs stored separately:

- `RAJA::sort_pairs< exec_policy >(keys_container, vals_container)`
- `RAJA::sort_pairs< exec_policy >(keys_container, vals_container, comparator)`

`RAJA::sort_pairs` methods generate the same output sequence of keys in `keys_container` as `RAJA::sort` does in `container` and reorders the sequence of values in `vals_container` by permuting the sequence of values in the same manner as the sequence of keys; i.e. the sequence of pairs is sorted based on comparing their keys.

Note: The comparator used in `RAJA::sort_pairs` only compares keys.

RAJA Stable Sorts

RAJA stable sorts are essentially the same as unstable sorts:

- `RAJA::stable_sort< exec_policy >(container)`
- `RAJA::stable_sort< exec_policy >(container, comparator)`

RAJA also provides stable sort pairs that operate on key-value pairs stored separately:

- `RAJA::stable_sort_pairs< exec_policy >(keys_container, vals_container)`
- `RAJA::stable_sort_pairs< exec_policy >(keys_container, vals_container, comparator)`

RAJA Comparison Operators

RAJA provides two operators that can be used to produce different ordered sorts:

- `RAJA::operators::less<T>`
- `RAJA::operators::greater<T>`

Note: All RAJA comparison operators are in the namespace `RAJA::operators`.

Sort Policies

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

Local Array

This section introduces RAJA *local arrays*. A `RAJA::LocalArray` is an array object with one or more dimensions 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> >;
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>
                  >
```

(continues on next page)

(continued from previous page)

```
                >
              >
            >;


//
// 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 RAJA permutations)
- Array dimensions

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

The kernel policy is a two-level nested loop policy (see *Complex Loops (RAJA::kernel)* for information about RAJA kernel policies) 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 loop 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.

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 on a CPU or use of GPU shared memory.

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::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 `statement::Tile` construct generates the outer loop over tiles and the `statement::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 tiling 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::tile_fixed<2>, RAJA::seq_exec,
      RAJA::statement::ForICount<0, RAJA::statement::Param<1>,
                                 RAJA::seq_exec,
        RAJA::statement::Lambda<0>
      >
    >
```

(continues on next page)

(continued from previous page)

```
  >;


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 lambda argument and the `statement::ForICount` type allows the local tile loop index to be accessed as a lambda argument. 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 which 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.

Plugins

RAJA supports user-made plugins that may be loaded either at compilation time (static plugins) or during runtime (dynamic plugins). These two methods are not mutually exclusive, as plugins loaded statically can be run alongside plugins that are loaded dynamically.

Using RAJA Plugins

Static vs Dynamic Loading

Static loading is done at compile time and requires recompilation in order to add, remove, or change a plugin. This is arguably the easier method to implement, requiring only simple file linking to make work. However, recompilation may get tedious and resource-heavy when working with many plugins or on large projects. In these cases, it may be better to load plugins dynamically, requiring no recompilation of the project most of the time.

Dynamic loading is done at runtime and only requires the recompilation or moving of plugin files in order to add, remove, or change a plugin. This will likely require more work to set up, but in the long run may save time and resources. RAJA checks the environment variable `RAJA_PLUGINS` for a path to a plugin or plugin directory, and automatically loads them at runtime. This means that a plugin can be added to a project as easily as making a shared object file and setting `RAJA_PLUGINS` to the appropriate path.

Quick Start Guide

Static Plugins

1. Build RAJA normally.
2. Either use an #include statement within the code or compiler flags to load your plugin file with your project at compile time. A brief example of this would be something like `g++ project.cpp plugin.cpp -lRAJA -fopenmp -ldl -o project`.
3. When you run your project, your plugin should work.

Dynamic Plugins

1. Build RAJA normally.

2. Compile your plugin to be a shared object file with a .so extension. A brief example of this would be something like `g++ plugin.cpp -lRAJA -fopenmp -fPIC -shared -o plugin.so`.

3. Set the environment variable RAJA_PLUGINS to be the path of your .so file. This can either be the path to its directory or to the shared object file itself. If the path is to a directory, it will attempt to load all .so files in that directory.

4. When you run your project, your plugins should work.

Interfacing with Plugins

The RAJA plugin API allows for limited interfacing between a project and a plugin. There are a couple of functions that allow for this to take place, `init_plugins` and `finalize_plugins`. These will call the corresponding `init` and `finalize` functions, respectively, of *every* currently loaded plugin. It's worth noting that plugins don't require either an init or finalize function by default.

- `RAJA::util::init_plugins();` - Will call the `init` function of every currently loaded plugin.
- `RAJA::util::init_plugins("path/to/plugins");` - Does the same as the above call to `init_plugins`, but will also dynamically load plugins located at the path specified.
- `RAJA::util::finalize_plugins();` - Will call the `finalize` function of every currently loaded plugin.

Creating Plugins For RAJA

Plugins are classes derived from the `RAJA::util::PluginStrategy` base class and implement the required functions for the API. An example implementation can be found at the bottom of this page.

Functions

The `preLaunch` and `postLaunch` functions are automatically called by RAJA before and after executing a kernel that uses `RAJA::forall` or `RAJA::kernel` methods.

- `void init(const PluginOptions& p) override {}` - runs on all plugins when a user calls `init_plugins`
- `void preCapture(const PluginContext& p) override {}` - is called before lambda capture in `RAJA::forall` or `RAJA::kernel`.
- `void postCapture(const PluginContext& p) override {}` - is called after lambda capture in `RAJA::forall` or `RAJA::kernel`.
- `void preLaunch(const PluginContext& p) override {}` - is called before `RAJA::forall` or `RAJA::kernel` runs a kernel.

- `void postLaunch(const PluginContext& p) override {}` - is called after `RAJA::forall` or `RAJA::kernel` runs a kernel.
- `void finalize() override {}` - Runs on all plugins when a user calls `finalize_plugins`. This will also unload all currently loaded plugins.

`init` and `finalize` are never called by RAJA by default and are only called when a user calls `RAJA::util::init_plugins()` or `RAJA::util::finalize_plugin()`, respectively.

Static Loading

If a plugin is to be loaded into a project at compile time, adding the following method call will add the plugin to the RAJA `PluginRegistry` and will be loaded every time the compiled executable is run. This requires the plugin to be loaded with either an `#include` statement within a project or with source code line such as:

```
static RAJA::util::PluginRegistry::add<PluginName> P("Name", "Description");
```

Dynamic Loading

If a plugin is to be dynamically loaded in a project at run time, the RAJA plugin API requires a few conditions to be met. The following must be true about the plugin, not necessarily of the project using it.

1. **The plugin must have the following factory function.** This will return a pointer to an instance of your plugin. Note that using `extern "C"` is required to search for the `getPlugin()` method call for the dynamically loaded plugin correctly:

   ```
   extern "C" RAJA::util::PluginStrategy *getPlugin ()
   {
     return new MyPluginName;
   }
   ```

2. **The plugin must be compiled to be a shared object with a .so extension.** A simple example containing required flags would be: `g++ plugin.cpp -lRAJA -fopenmp -fPIC -shared -o plugin.so`.

   At the moment, RAJA will only attempt to load files with .so extensions. It's worth noting why these flags (or their equivalents) are important.

   - `-lRAJA -fopenmp` are standard flags for compiling the RAJA library.
   - `-fPIC` tells the compiler to produce *position independent code*, which prevents conflicts in the address space of the executable.
   - `-shared` will let the compiler know that you want the resulting object file to be shared, removing the need for a *main* as well as giving dynamically loaded executables access to functions flagged with `extern "C"`.

3. **The** `RAJA_PLUGINS` **environment variable has been set**, or a user has made a call to `RAJA::util::init_plugins("path");` with a path specified to either a directory or a .so file. It's worth noting that these are not mutually exclusive. RAJA will look for plugins based on the environment variable on program startup and new plugins may be loaded after that by calling the `init_plugins()` method.

Example Plugin Implementation

The following is an example plugin that simply will print out the number of times a kernel has been launched and has the ability to be loaded either statically or dynamically.

```
#include "RAJA/util/PluginStrategy.hpp"

#include <iostream>

class CounterPlugin :
  public RAJA::util::PluginStrategy
{
  public:
  void preCapture(const RAJA::util::PluginContext& p) override {
    if (p.platform == RAJA::Platform::host)
    {
      std::cout << " [CounterPlugin]: Capturing host kernel for the " << ++host_
↪capture_counter << " time!" << std::endl;
    }
    else
    {
      std::cout << " [CounterPlugin]: Capturing device kernel for the " << ++device_
↪capture_counter << " time!" << std::endl;
    }
  }

  void preLaunch(const RAJA::util::PluginContext& p) override {
    if (p.platform == RAJA::Platform::host)
    {
      std::cout << " [CounterPlugin]: Launching host kernel for the " << ++host_
↪launch_counter << " time!" << std::endl;
    }
    else
    {
      std::cout << " [CounterPlugin]: Launching device kernel for the " << ++device_
↪launch_counter << " time!" << std::endl;
    }
  }

  private:
   int host_capture_counter;
   int device_capture_counter;
   int host_launch_counter;
   int device_launch_counter;
};

// Statically loading plugin.
static RAJA::util::PluginRegistry::add<CounterPlugin> P("Counter", "Counts number of␣
↪kernel launches.");

// Dynamically loading plugin.
extern "C" RAJA::util::PluginStrategy *getPlugin ()
{
  return new CounterPlugin;
}
```

CHAI Plugin

RAJA provides abstractions for parallel execution, but does not support a memory model for managing data in heterogeneous memory spaces. The CHAI library provides an array abstraction that integrates with RAJA to enable automatic copying of data at runtime to the proper execution memory space for a RAJA-based kernel based on the

RAJA exection policy used to execute the kernel. Then, the data can be accessed inside the kernel as needed.

To build CHAI with RAJA integration, you need to download and install CHAI with the `ENABLE_RAJA_PLUGIN` option turned on. Please see the CHAI project for details.

After CHAI has been built with RAJA support enabled, applications can use CHAI `ManangedArray` objects to access data inside a RAJA kernel. 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 since it is used in a kernel that will run with a RAJA CUDA execution policy. So it copies the data from CPU to GPU, making it available for access in the RAJA kernel. Next, it is printed in the second kernel which runs on the CPU (indicated by the RAJA sequential execution policy). So CHAI copies the data back to the host CPU. All necessary data copies are done transparently on demand for each kernel.

WorkGroup

In this section, we describe the basics of RAJA workgroups. `RAJA::WorkPool`, `RAJA::WorkGroup`, and `RAJA::WorkSite` class templates comprise the RAJA interface for grouped loop execution. `RAJA::WorkPool` takes a set of simple loops (e.g., non-nested loops) and instantiates a `RAJA::WorkGroup`. `RAJA::WorkGroup` represents an executable form of those loops and when run makes a `RAJA::WorkSite`. `RAJA::WorkSite` holds all of the resources used for a single run of the loops. Be aware that the RAJA workgroup constructs API is still being developed and may change in later RAJA releases.

Note:

- All **workgroup** constructs are in the namespace `RAJA`.
- The `RAJA::WorkPool`, `RAJA::WorkGroup`, and `RAJA::WorkSite` class templates are templated on:
  - **a WorkGroup policy which is composed of:**
    - a work execution policy.
    - a work ordering policy.
    - a work storage policy.
  - an index type that is the first argument to the loop bodies.
  - a list of extra argument types that are the rest of the arguments to the loop bodies.
  - an allocator type to be used for the memory used to store and manage the loop bodies.
- **The `RAJA::WorkPool::enqueue` method takes two arguments:**
  - an iteration space object, and
  - a lambda expression representing the loop body.

- Multi-dimensional loops can be used with RAJA::CombiningAdapter see, *MultiDimensional loops using Simple loop APIs (RAJA::CombiningAdapter)*.

Examples showing how to use RAJA workgroup methods may be found in the *RAJA Tutorial*.

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

Policies

The behavior of the RAJA workgroup constructs is determined by a policy. The RAJA::WorkGroupPolicy has three components, a work execution policy, a work ordering policy, and a work storage policy. RAJA::WorkPool, RAJA::WorkGroup, and RAJA::WorkSite class templates all take the same policy and template arguments. For example:

```
using workgroup_policy = RAJA::WorkGroupPolicy <
                             RAJA::seq_work,
                             RAJA::ordered,
                             RAJA::ragged_array_of_objects >;
```

is a workgroup policy that will run loops sequentially on the host in the order they were enqueued and store the loop bodies sequentially in single buffer in memory.

The work execution policy acts like the execution policies used with RAJA::forall and determines the backend used to run the loops and the parallelism within each loop.

Work Execution Policies	Brief description
seq_work	Execute loop iterations strictly sequentially.
simd_work	Execute loop iterations sequentially and try to force generation of SIMD instructions via compiler hints in RAJA internal implementation.
loop_work	Execute loop iterations sequentially and allow compiler to generate any optimizations.
omp_work	Execute loop iterations in parallel using OpenMP.
tbb_work	Execute loop iterations in parallel using TBB.
cuda_work<BLOCK_SIZE>,	Execute loop iterations in parallel
cuda_work_async<BLOCK_SIZE>	using a CUDA kernel launched with given thread-block size.
omp_target_work	Execute loop iterations in parallel using OpenMP target.

The work ordering policy acts like the segment iteration execution policies when RAJA::forall is used with a RAJA::IndexSet and determines the backend used when iterating over the loops and the parallelism between each loop.

The work storage policy determines the strategy used to allocate and layout the storage used to store the ranges, loop bodies, and other data necessary to implement the workstorage constructs.

Work Storage Policies	Brief description
array_of_pointers	Store loop data in individual allocations and keep an array of pointers to the individual loop data allocations.
ragged_array_of_objects	Stores loops sequentially in a single allocation, reallocating and moving the loop data items as needed, and keep an array of offsets to the individual loop data items.
constant_stride_array_of_objects	Store loops sequentially in a single allocation with a consistent stride between loop data items, reallocating and/or changing the stride and moving the loop data items as needed.

Arguments

The next two template arguments to the workgroup constructs determine the call signature of the loop bodies that may be added to the workgroup. The first is an index type which is the first parameter in the call signature. Next is a list of types called `RAJA::xargs`, short for extra arguments, that gives the rest of the types of the parameters in the call signature. The values of the extra arguments are passed in when the loops are run, see *WorkGroup*. For example:

```
int, RAJA::xargs<>
```

can be used with lambdas with the following signature:

```
[=](int) { ... }
```

and:

```
int, RAJA::xargs<int*, double>
```

can be used with lambdas with the following signature:

```
[=](int, int*, double) { ... }
```

Allocators

The last template argument to the workgroup constructs is an allocator type that conforms to the allocator named requirement used in the standard library. This gives you control over how memory is allocated, for example with umpire, and what memory space is used, both of which have poerformance implications. Find the requirements for allocator types along with a simple example here https://en.cppreference.com/w/cpp/named_req/Allocator. The default allocator used by the standard template library may be used with ordered and non-GPU policies:

```
using Allocator = std::allocator<char>;
```

Note:

- The allocator type must use template argument char.
- **Allocators must provide memory that is accessible where it is used.**
  - Ordered work order policies only require memory that is accessible where loop bodies are enqueued.
  - Unordered work order policies require memory that is accessible from both where the loop bodies are enqueued and from where the loop is executed based on the work execution policy.

- For example when using cuda work exeution policies with cuda unordered work order policies pinned memory is a good choice because it is always accessible on the host and device.

WorkPool

The `RAJA::WorkPool` class template holds a set of simple (e.g., non-nested) loops that are enqueued one at a time. Note that simple multi-dimensional loops can be adapted into simple loops via `RAJA::CombiningAdapter`, see *MultiDimensional loops using Simple loop APIs (RAJA::CombiningAdapter)*. For example, to enqueue a C-style loop that adds two vectors, like:

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

is as simple as calling enqueue on a `RAJA::WorkPool` object and passing the same arguments you would pass to `RAJA::forall`.:

```
using WorkPool_type = RAJA::WorkPool< workgroup_policy,
                                      int, RAJA::xargs<>,
                                      Allocator >;
WorkPool_type workpool(Allocator{});

workpool.enqueue(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});
```

Note that WorkPool may have to allocate and reallocate multiple times to store a set of loops depending on the work storage policy. Reallocation can be avoided by reserving enough memory before adding any loops.:

```
workpool.reserve(num_loops, storage_bytes);
```

Here `num_loops` is the number of loops to allocate space for and `num_storage_bytes` is the amount of storage to allocate. These may be used differently depending on the work storage policy. The number of loops enqueued in a `RAJA::WorkPool` and the amount of storage used may be queried using:

```
size_t num_loops     = workpool.num_loops();
size_t storage_bytes = workpool.storage_bytes();
```

Storage will automatically reserved when reusing a *RAJA::WorkPool‘* object based on the maximum seen values for num_loops and storage_bytes.

When you've added all the loops you want to the set, you can call instantiate on the `RAJA::WorkPool` to generate a `RAJA::WorkGroup`.:

```
WorkGroup_type workgroup = workpool.instantiate();
```

WorkGroup

The `RAJA::WorkGroup` class template is responsible for hanging onto the set of loops and running the loops. The `RAJA::WorkGroup` owns its loops and must not be destroyed before any loops run asynchronously using it have completed. It is instantiated from a `RAJA::WorkPool` object which transfers ownership of a set of loops to the `RAJA::WorkGroup` and prepares the loops to be run. For example:

```
using WorkGroup_type = RAJA::WorkGroup< workgroup_policy,
                                        int, RAJA::xargs<>,
                                        Allocator >;
WorkGroup_type workgroup = workpool.instantiate();
```

creates a `RAJA::WorkGroup` `workgroup` from the loops in `workpool` and leaves `workpool` empty and ready for reuse. When you want to run the loops simply call run on `workgroup` and pass in the extra arguments:

```
WorkSite_type worksite = workgroup.run();
```

In this case no extra arguments were passed to run because the `RAJA::WorkGroup` specified no extra arguments `RAJA::xargs<>`. Passing extra arguments when the loops are run lets you delay creation of those arguments until you plan to run the loops. This lets the value of the arguments depend on the loops in the set. A simple example of this may be found in the tutorial here *RAJA Tutorial*. Run produces a `RAJA::WorkSite` object.

WorkSite

The `RAJA::WorkSite` class template is responsible for extending the lifespan of objects used when running loops asynchronously. This means that the `RAJA::WorkSite` object must remain alive until the call to run has been synchronized. For example the scoping here:

```
{
  using WorkSite_type = RAJA::WorkSite< workgroup_policy,
                                        int, RAJA::xargs<>,
                                        Allocator >;
  WorkSite_type worksite = workgroup.run();

  // do other things

  synchronize();
}
```

ensures that `worksite` survives until after synchronize is called.

Vectorization (SIMD/SIMT)

Warning: This section describes an initial draft of an incomplete, experimental RAJA capability. It is not considered ready for production. A basic description is provided here so that (potentially) interested users can take a look, try it out, and provide input if they wish to do so.

The RAJA team is experimenting with an API for SIMD/SIMT programming. The goal is to make the implementation perform as well as if one used vectorization intrinsics directly in their code, but without the software complexity and maintenance burden associated with doing that. In particular, our goal is to *guarantee* that specified vectorization occurs without needing to explicitly use intrinsics in user code or rely on compiler auto-vectorization implementations.

Note: All RAJA vectorization types are in the namespace `RAJA::expt`.

Currently, the main abstractions developed in RAJA so far are:

- `Register` wraps underlying SIMD/SIMT hardware registers and provides consistent uniform access to them, using intrinsics under the API when possible. The RAJA register abstraction currently supports the following hardware-specific ISAs : AVX, AVX2, AVX512, CUDA, and HIP.
- `Vector` builds on `Register` to provide arbitrary length vectors and operations on them.
- `Matrix` builds on `Register` to provide arbitrary-sized matrices, column-major and row-major layouts, and operations on them.

Finally, these capabilities integrate with RAJA *View and Layout* capabilities, which implements am expression-template system that allows a user to write linear algebra expressions on arbitrarily sized scalars, vectors, and matrices and have the appropriate SIMD/SIMT instructions performed during expression evaluation.

Why Are We Doing This?

Quoting Tim Foley in Matt Pharr's blog: "Auto-vectorization is not a programming model". Unless, of course, you consider "hope for the best" to be a sound plan.

Auto-vectorization is problematic for multiple reasons. First, vectorization is not explicit in the source code and so compilers must divine correctness when attempting to apply vectorization optimizations. Since most compilers are very conservative in this regard, many vectorization opportunities are typically missed when one relies solely on compiler auto-vectorization. Second, every compiler will treat your code differently since compiler implementations use different heuristics, even for different versions of the same compiler. So performance portability is not just an issue with respect to hardware, but also across compilers. Third, it is impossible in general for most application developers to clearly understand the decisions made by a compiler during its optimization process.

Using vectorization intrinsics in application source code is also problematic because different processors support different instruction set architectures (ISAs) and so source code portability requires a mechanism that insulates it from architecture-specific code.

GPU programming makes us be explicit about parallelization, and SIMD is really no different. RAJA enables single-source portable code across a variety of programming model back-ends. The RAJA vectorization abstractions introduced here are an attempt to bring a level of convergence between SIMD and GPU programming by providing uniform access to hardware-specific acceleration.

Note: **Auto-vectorization is not a programming model.** –Tim Foley

Register

`RAJA::expt::Register<T, REGISTER_POLICY>` is a class template that takes a a data type parameter `T` and a register policy `REGISTER_POLICY` that indicates the hardware register type. The `RAJA::expt::Register` interface provides uniform access to register-level operations. It is intended as a building block for higher level abstractions. A `RAJA::expt::Register` type represents one SIMD register on a CPU architecture and 1 value/SIMT lane on a GPU architecture.

Note: A user can use the `RAJA::expt::Register` type directly in their code. However, we do not recommend this. Instead, we want users to employ higher level abstractions that RAJA provides.

`RAJA::expt::Register` supports four scalar element types, `int32_t`, `int64_t`, `float`, and `double`. These are the only types that are portable across all SIMD/SIMT architectures. `Bfloat`, for example, is not portable, so we don't provide support for that type.

`RAJA::expt::Register` supports the following SIMD/SIMT hardware-specific ISAs: AVX, AVX2, and AVX512 for SIMD CPU vectorization, and CUDA warp, HIP wavefront for GPUs. Scalar support is provided for all hardware for portability and experimentation/analysis. Extensions to support other architectures may be forthcoming and should be straightforward to implement.

Register Operations

`RAJA::expt::Register` provides various operations, including:

- Basic SIMD handling: get element, broadcast
- Memory operations: load (packed, strided, gather) and store (packed, strided, scatter)
- SIMD element-wise arithmetic: add, subtract, multiply, divide, vmin, vmax
- Reductions: dot-product, sum, min, max
- Special operations for matrix operations: permutations, segmented operations

Register DAXPY Example

The following is a code example that shows using the `RAJA::expt::Register` class to perform a DAXPY kernel with AVX2 CPU SIMD instructions. Again, we do not recommend that you write code directly using the Register class, but use the higher level VectorRegister abstraction. However, this example demonstrates how the higher level abstractions are using the Register class:

```
// define array length
int len = ...;

// data used in kernel
double a = ...;
double const *X = ...;
double const *Y = ...;
double *Z = ...;

using reg_t = RAJA::expt::Register<double, RAJA::expt::avx2_register>;
int reg_width = reg_t::s_num_elem;    // width of avx2 register is 4 doubles

// Compute daxpy in chunks of 4 values at one time
for (int i = 0;i < len; i += reg_width){
  reg_t x, y;

  // load 4 consecutive values of X, Y arrays into registers
  x.load_packed( X+i );
  y.load_packed( Y+i );

  // perform daxpy on 4 values simultaneously (store in register)
  reg_t z = a * x + y;

  // store register result in Z array
  z.store_packed( Z+i );
}

// loop postamble code
int remainder = len % reg_width;
if (remainder) {
```

(continues on next page)

(continued from previous page)

```
  reg_t x, y;

  // 'i' is the starting array index of the remainder
  int i = len - remainder;

  // load remainder values of X, Y arrays into registers
  x.load_packed_n( X+i, remainder );
  y.load_packed_n( Y+i, remainder );

  // perform daxpy on remainder values simultaneously (store in register)
  reg_t z = a * x + y;

  // store register result in Z array
  z.store_packed_n(Z+i, remainder);
}
```

This code is guaranteed to vectorize since the `RAJA::expt::Register` operations insert the appropriate SIMD intrinsic operations into the method calls. Note that `RAJA::expt::Register` provides overloads of basic arithmetic operations so that the DAXPY operation itself (z = a * x + y) looks like vanilla scalar code.

Note that since we are using bare pointers to the data, load and store operations are performed by explicit method calls in the code. Also, we must write (duplicate) postamble code to handle cases where the array length (len) is not an integer multiple of the register width. The postamble code perform the DAXPY operation on the *remainder* of the array that remains after the for-loop.

These extra lines of code should make it clear why we do not recommend using ‘‘RAJA::Register‘‘ directly in application code.

Tensor Register

`RAJA::expt::TensorRegister< >` is a class template that provides a higher-level interface on top of the `RAJA::expt::Register` class. `RAJA::expt::TensorRegister< >` wraps one or more `RAJA::expt::Register< >` objects to create a tensor-like object.

Note: As with `RAJA::expt::Register`, we don’t recommend using `RAJA::expt::TensorRegister` directly. Rather, we recommend using use-case specific types that RAJA provides and which are described below.

To make code cleaner and more readable, the specific types are intended to be used with ‘‘RAJA::View‘‘ and ‘‘RAJA::expt::TensorIndex‘‘ objects.

Vector Register

`RAJA::expt::VectorRegister<T, REGISTER_POLICY, NUM_ELEM>` provides an abstraction for a vector of arbitrary length. It is implemented using one or more `RAJA::expt::Register` objects. The vector length is independent of the underlying register width. The template parameters are: `T` data type, `REGISTER_POLICY` vector register policy, and `NUM_ELEM` number of data elements of type `T` that fit in a register. The last two of these have defaults for all cases, so they do not usually need to be provided by a user.

Earlier, we said that we do not recommended using `RAJA::expt::Register` directly. The reason for this is that it is good to decouple vector length from hardware register size since it allows one to write simpler, more readable code that is easier to get correct. This should be clear from the code example below.

Vector Register DAXPY Example

The following code example shows the DAXPY computation shown above written using `RAJA::expt::VectorRegister`, `RAJA::expt::VectorIndex`, and `RAJA::View` classes, which obviate the need for the extra lines of code discussed earlier:

```
// define array length and data used in kernel (as before)
int len = ...;
double a = ...;
double const *X = ...;
double const *Y = ...;
double *Z = ...;

// define vector register and index types
using vec_t = RAJA::expt::VectorRegister<double, RAJA::expt::avx2_register>;
using idx_t = RAJA::expt::VectorIndex<int, vec_t>;

// wrap array pointers in RAJA View objects
auto vX = RAJA::make_view( X, len );
auto vY = RAJA::make_view( Y, len );
auto vZ = RAJA::make_view( Z, len );

// 'all' knows the length of vX, vY, and vZ from the View objects
// and it encodes the vector type
auto all = idx_t::all();

// compute the complete array daxpy in one line of code
// this produces a vectorized loop, and the loop postamble
vZ( all ) = a * vX( all ) + vY( all );
```

This code has several advantages over the previous example. It is guaranteed to vectorize and is much easier to read, get correct, and maintain since the `RAJA::View` class handles the looping and postamble code automatically to allow arrays of arbitrary size. The `RAJA::View` class provides overloads of the arithmetic operations based on the 'all' type and inserts the appropriate SIMD instructions and load/store operations to vectorize the operations as in the earlier example. It may be considered by some to be inconvenient to have to use the `RAJA::View` class, but it is easy to wrap bare pointers as can is shown in the example.

Expression Templates

The figure below shows the sequence of SIMD operations, in the form of an *abstract syntax tree (AST)*, applied in the DAXPY code by the RAJA constructs used in the code example. During compilation, a tree of *expression template* objects is constructed based on the order of operations that appear in the kernel. Specifically, the operation sequence is the following:

1. Load a chunk of values in 'vX' into a register.
2. Broadcast the scalar value 'a' to each slot in a vector register.
3. Load a chunk of values in 'vY' into a register.
4. Multiply values in the 'a' register and 'vX' register and multiply by the values in the 'vY' register in a single vector FMA (Fused Multiply-Add) operation, storing the result in a register.
5. Write the result in the register to the 'vZ' array.

`RAJA::View` objects indexed by `RAJA::TensorIndex` objects (`RAJA::VectorIndex` in this case) return *LoadStore* expression template objects. Each expression template object is evaluated on assignment and a register chunk size of values is loaded into another register object. Finally, the left-hand side of the expression is evaluated by storing the chunk of values in the right-hand side result register into the array on the left-hand side of the equal sign.

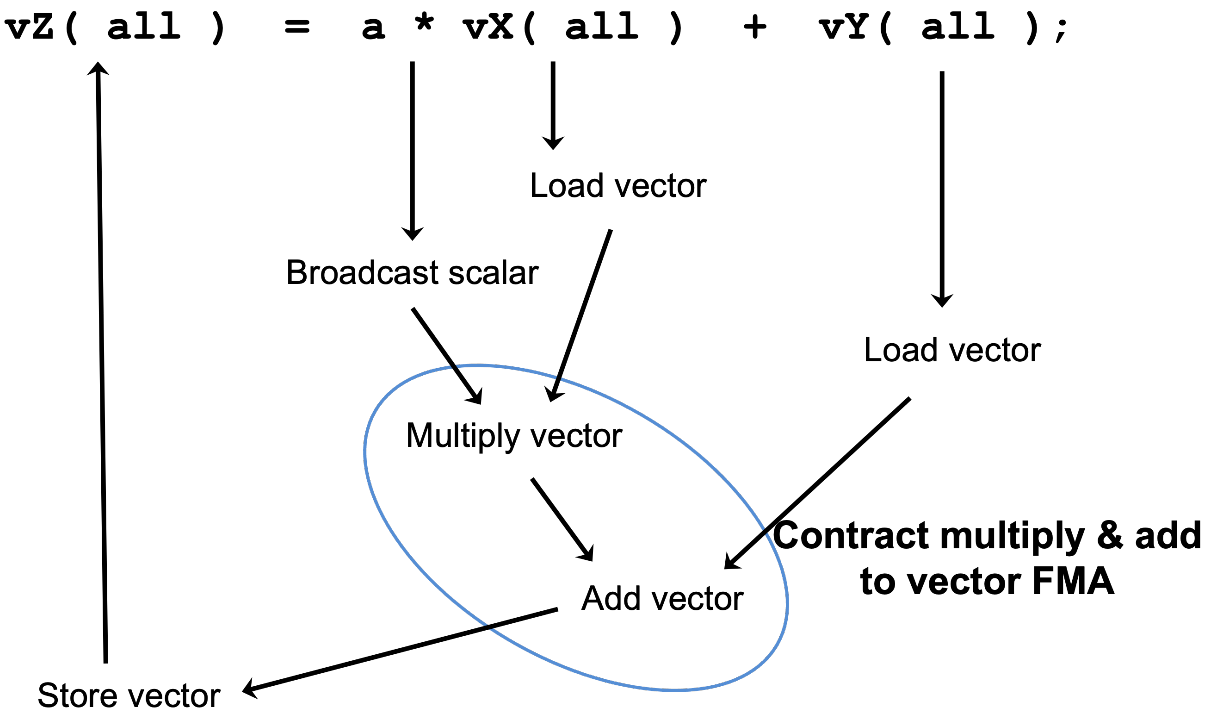
vZ(all) = a * vX(all) + vY(all);
Load vector
Broadcast scalar
Load vector
Multiply vector
Contract multiply & add to vector FMA
Add vector
Store vector

Fig. 5: An AST illustration of the SIMD operations in the DAXPY code.

CPU/GPU Portability

It is important to note that the code in the example in the previous section is *not* portable to run on a GPU because it does not include a way to launch a GPU kernel. The following code example shows how to enable the code to run on either a CPU or GPU via a run time choice:

```
// array lengths and data used in kernel same as above

// define vector register and index types
using vec_t = RAJA::expt::VectorRegister<double>;
using idx_t = RAJA::expt::VectorIndex<int, vec_t>;

// array pointers wrapped in RAJA View objects as before
// ...

using cpu_launch = RAJA::expt::seq_launch_t;
using gpu_launch = RAJA::expt::cuda_launch_t<false>; // false => launch
                                                      // CUDA kernel
                                                      // synchronously

using pol_t =
  RAJA::expt::LoopPolicy< cpu_launch, gpu_launch >;

RAJA::expt::ExecPlace cpu_or_gpu = ...;
```

(continues on next page)

(continued from previous page)

```
RAJA::expt::launch<pol_t>( cpu_or_gpu, resources,

                           [=] RAJA_HOST_DEVICE (context ctx) {
                               auto all = idx_t::all();
                               vZ( all ) = a * vX( all ) + vY( all );
                           }
                         );
```

This version of the kernel can be run on a CPU or GPU depending on the run time chosen value of the variable `cpu_or_gpu`. When compiled, the code will generate versions of the kernel for the CPU and GPU based on the parameters in the `pol_t` loop policy. The CPU version will be the same as the version in the previous section. The GPU version is essentially the same but will run in a GPU kernel. Note that there is only one template argument passed to the register when `vec_t` is defined. `RAJA::expt::VectorRegister<double>` uses defaults for the register policy, based on the system hardware, and number of data elements of type double that will fit in a register.

Matrix Registers

RAJA provides `RAJA::expt::TensorRegister` type aliases to support matrices of arbitrary size and shape. These are:

- `RAJA::expt::SquaretMatrixRegister<T, LAYOUT, REGISTER_POLICY>` which abstracts operations on an N x N square matrix.
- **`RAJA::expt::RectMatrixRegister<T, LAYOUT, ROWS, COLS, REGISTER_POLICY>`**
  which abstracts operations on an N x M rectangular matrix.

Matrices are implemented using one or more `RAJA::expt::Register` objects. Data layout can be row-major or column major. Matrices are intended to be used with `RAJA::View` and `RAJA::expt::TensorIndex` objects, similar to what was shown above with `RAJA::expt::VectorRegister` example.

Matrix operations support matrix-matrix, matrix-vector, and vector-matrix multiplication, and transpose operations. Rows or columns can be represented with one or more registers, or a power-of-two fraction of a single register. This is important for CUDA GPU warp/wavefront registers, which are 32-wide for CUDA and 64-wide for HIP.

Here is a simple code example that performs the matrix-analogue of the vector DAXPY operation presented above using square matrices:

```
// define matrix size and data used in kernel (similar to before)
int N = ...;
double a = ...;
double const *X = ...;
double const *Y = ...;
double *Z = ...;

// define matrix register and row/column index types
using mat_t = RAJA::expt::SquareMatrixRegister<double,
                                               RAJA::expt::RowMajorLayout>;
using row_t = RAJA::expt::RowIndex<int, mat_t>;
using col_t = RAJA::expt::ColIndex<int, mat_t>;

// wrap array pointers in RAJA View objects (similar to before)
auto mX = RAJA::make_view( X, N, N );
auto mY = RAJA::make_view( Y, N, N );
auto mZ = RAJA::make_view( Z, N, N );
```

(continues on next page)

(continued from previous page)

```
using cpu_launch = RAJA::expt::seq_launch_t;
using gpu_launch = RAJA::expt::cuda_launch_t<false>; // false => launch
                                                     // CUDA kernel
                                                     // synchronously
using pol_t =
  RAJA::expt::LoopPolicy< cpu_launch, gpu_launch >;

RAJA::expt::ExecPlace cpu_or_gpu = ...;

RAJA::expt::launch<pol_t>( cpu_or_gpu, resources,

    [=] RAJA_HOST_DEVICE (context ctx) {
       auto rows = row_t::all();
       auto cols = col_t::all();
       mZ( rows, cols ) = a * mX( rows, cols ) + mY( rows, cols );
    }
  );
```

Conceptually, as well as implementation-wise, this is similar to the previous vector example except the operations are in two dimensions. The kernel code is easy to read, it is guaranteed to vectorize, and iterating over the data is handled by RAJA (register width sized chunk, plus postamble scalar operations). Again, the `RAJA::View` arithmetic operation overloads insert the appropriate vector instructions in the code.

6.1.3 Application Considerations

> **Warning: Comming soon!! Stay tuned.**

6.1.4 RAJA Tutorial

In addition to the tutorial portion of this RAJA User Guide, we maintain a repository of tutorial presentation materials here RAJA Tutorials Repo.

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. The RAJA tutorial evolves as we add new features to RAJA, so refer to it periodically if you are interested in learning about them.

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 how RAJA use employs C++ templates and lambda expressions, which is 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. This is by design as developers of many of applications that use RAJA prefer to manage memory themselves. 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 unified memory (UM), if available. RAJA developers also support a library called CHAI which complements RAJA by providing a alternative to manual host-device memory copy calls or UM. For more information, see *Plugins*.

A Little C++ Background

RAJA makes heavy use of C++ templates and using RAJA most easily and effectively is done by representing the bodies of loop kernels as C++ lambda expressions. Alternatively, C++ functors can be used, but they make application source code more complex, potentially placing a significant negative burden on source code readability and maintainability.

C++ Templates

C++ templates enable one to write generic code and have the compiler generate a specific implementation for each set of template parameter types you use. For example, the `RAJA::forall` method to execute loop kernels is a template method defined as:

```
template <typename ExecPol,
          typename IdxType,
          typename LoopBody>
forall(IdxType&& idx, LoopBody&& body) {
   ...
}
```

Here, "ExecPol", "IdxType", and "LoopBody" are C++ types a user specifies in their code; for example:

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

The "IdxType" and "LoopBody" types are deduced by the compiler based on what arguments are passed to the `RAJA::forall` method. Here, the loop body type is defined by the lambda expression:

```
[=](int i) { a[i] = b[i] + c[i]; }
```

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; specifically, a *closure* that stores a function with a data environment. That is, 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 in a regular 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; e.g., loop iteration 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; i.e., *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; };      // error: all lambda arguments captured by value,
                      //        so cannot assign to 'x'.
[x, &y](){ x = y; };  // error: cannot assign to 'x' since it is captured
                      //        by value.
```

Specifically, a variable hat is captured by value is read-only.

A Few Notes About Lambda Usage With RAJA

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<BLOCK_SIZE>>( 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.

- **Use ‘host-device’ annotation on a lambda carefully.**

  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 be applied carefully 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 may run noticeably slower on a CPU than the same lambda with no annotation depending on the version of the nvcc compiler you are using.

- **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<BLOCK_SIZE>>( range, [=] __device__ (int i) {
  // use ref_to_global_val
} );
```

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

  Although this is inconsistent with the C++ standard (local stack arrays are properly captured in lambdas for code that will execute on a CPU), attempting to access elements in a local stack array in a CUDA device lambda may generate a compilation error depending on the version of the nvcc compiler you are using. 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<BLOCK_SIZE>>(range, [=] __device__ (int i) {
  // access entries of bounds.array
} );
```

  This issue appears to be resolved in in the 10.1 release of CUDA. If you are using an earlier version of nvcc, an implementation similar to the one above will be required.

RAJA Examples

The remainder of this tutorial illustrates how to use RAJA features with 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 multithreading) and CUDA GPU execution. Examples that show how to use RAJA with other parallel programming model back-ends that are in development will appear in future RAJA releases. 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.

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, scans, and sorts.

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 multithreaded 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.

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.

For performance tuning, the `RAJA::cuda_exec_explicit` policy is also provided. This allows the user to specify the number of blocks allocated per streaming multiprocessor (SM) to allow additional block level parallelism. Note that the third boolean argument representing asynchronous execution can be omitted, and is `false` by default:

```
RAJA::forall<RAJA::cuda_exec_explicit<CUDA_BLOCK_SIZE, 2, Asynchronous>>
↪(RAJA::RangeSegment(0, N),
  [=] RAJA_DEVICE (int i) {
  c[i] = a[i] + b[i];
});
```

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.

Similarly, to run the kernel on a GPU using the RAJA HIP back-end, we use the `RAJA::hip_exec` policy:

```
RAJA::forall<RAJA::hip_exec<HIP_BLOCK_SIZE>>(RAJA::RangeSegment(0, N),
  [=] RAJA_DEVICE (int i) {
  d_c[i] = d_a[i] + d_b[i];
});
```

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) {
```

(continues on next page)

(continued from previous page)

```
    dot += a[i] * b[i];
  }
```

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:

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 multithreaded 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();
```

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

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();
```

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.

Similarly, for the RAJA HIP variant:

```
  RAJA::ReduceSum<RAJA::hip_reduce, double> hpdot(0.0);

  RAJA::forall<RAJA::hip_exec<HIP_BLOCK_SIZE>>(RAJA::RangeSegment(0, N),
    [=] RAJA_DEVICE (int i) {
```

(continues on next page)

(continued from previous page)

```
    hpdot += d_a[i] * d_b[i];
  });

  dot = hpdot.get();
```

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 in a standard vector, create a list segment from that, 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(), host_res );

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

Note that we are using the following type aliases:

```
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 understand what happens when using list segments. 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 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(), host_res );

  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;
  });
```

The fact that RAJA always passes loop index values to lambdas in a kernel explains why we can run a kernel with multiple segment types in a single RAJA construct as we discuss next.

RAJA IndexSets

The `RAJA::TypedIndexSet` template is a container that can hold any number of segments, of the same or different types. 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 list segment we created earlier 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 wondering: What is the 'SEQ_ISET_EXECPOL' type used for the execution policy?

Well, it is similar to 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:

```
//
// Collect indices in a vector to create list segment
//
  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(), host_res );

  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) {
    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 code by iterating over the segments sequentially and executing each segment in parallel using OpenMP multithreading, 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 the following policy:

```
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 using either CUDA or HIP kernel, we would use a policy, such as:

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

or:

```
using HIP_ISET_EXECPOL = RAJA::ExecPolicy<RAJA::seq_segit,
                                          RAJA::hip_exec<HIP_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.

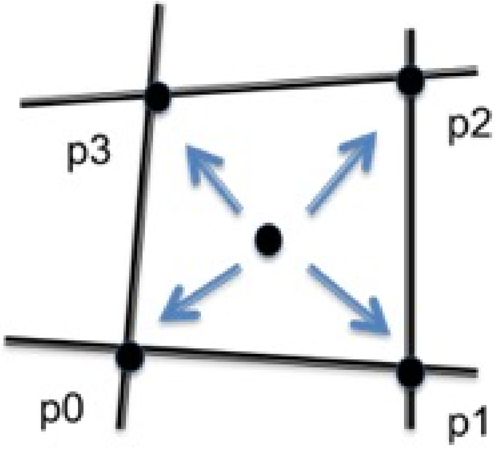

Fig. 6: A portion of the area of each mesh element is summed to the vertices surrounding the element.

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 ;
    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;

camp::resources::Resource host_res{camp::resources::Host()};

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

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

```
using EXEC_POL2 = RAJA::ExecPolicy<RAJA::seq_segit,
                                   RAJA::seq_exec>;

RAJA::forall<EXEC_POL2>(colorset, [=](int ie) {
  int* iv = &(elem2vert_map[4*ie]);
  vertexvol[ iv[0] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[1] ] += elemvol[ie] / 4.0 ;
```

(continues on next page)

(continued from previous page)

```
    vertexvol[ iv[2] ] += elemvol[ie] / 4.0 ;
    vertexvol[ iv[3] ] += elemvol[ie] / 4.0 ;
  });
```

Note that 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_cuda, [=] 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 RAJA HIP variant is similar.

The file `RAJA/examples/tut_vertexsum-coloring.cpp` contains a complete working example code, including a RAJA HIP variant.

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;
```

(continues on next page)

(continued from previous page)

```
//
// 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:

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];

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

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

(continues on next page)

(continued from previous page)

```
});

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 multithreading 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 GPU using CUDA policies:

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

or HIP policies:

```
using EXEC_POL3    = RAJA::hip_exec<HIP_BLOCK_SIZE>;
using REDUCE_POL3 = RAJA::hip_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:

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

  RAJA::atomicAdd<RAJA::omp_atomic>(&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::atomicAdd` operation uses an OpenMP atomic policy, which is compatible with the OpenMP loop execution policy.

The CUDA and HIP versions are similar:

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

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

});
```

and:

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

  RAJA::atomicAdd<RAJA::hip_atomic>(&d_bins[d_array[i]], 1);

});
```

Here, the atomic add operations uses CUDA and HIP atomic policies, which are compatible with the CUDA and HIP loop execution policies.

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

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

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

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

});
```

and the HIP version:

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

    RAJA::atomicAdd<RAJA::auto_atomic>(&d_bins[d_array[i]], 1);

  });
```

The same CUDA and HIP loop execution policies as in the previous examples are used.

The file `RAJA/examples/tut_atomic-histogram.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 initialize vector data
//
  int* in = memoryManager::allocate<int>(N);
  int* out = memoryManager::allocate<int>(N);

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

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>(RAJA::make_span(in, N),
                                     RAJA::make_span(out, N));
```

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

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

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

```
RAJA::inclusive_scan<RAJA::omp_parallel_for_exec>(RAJA::make_span(in, N),
                                                  RAJA::make_span(out, N),
                                                  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>(RAJA::make_span(in, N),
                                     RAJA::make_span(out, N),
                                     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>>(RAJA::make_span(in, N),
                                                       RAJA::make_span(out, N),
                                                       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:

```
  RAJA::inclusive_scan_inplace<RAJA::seq_exec>(RAJA::make_span(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 -1
```

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

```
  RAJA::exclusive_scan_inplace<RAJA::seq_exec>(RAJA::make_span(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:

```
  RAJA::exclusive_scan_inplace<RAJA::omp_parallel_for_exec>(RAJA::make_span(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
```

and the only difference is the execution policy template parameter.

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

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

Note: RAJA scans for the HIP back-end are similar to those for CUDA.

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

Parallel Sort Operations

Key RAJA features shown in this section:

- `RAJA::sort` operation
- `RAJA::sort_pairs` operation
- `RAJA::stable_sort` operation
- `RAJA::stable_sort_pairs` operation
- RAJA comparators for different types of sorts; e.g., less, greater

Below, we present examples of RAJA sequential, OpenMP, and CUDA sort operations and show how different sort orderings can be achieved by passing different RAJA comparators to the RAJA sort template methods. Each comparator is a template type, where the template argument is the type of the values it compares. For a summary of RAJA sort functionality, please see *Parallel Sort Operations*.

Note: RAJA sort 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 initialize vector data
//
  int* in = memoryManager::allocate<int>(N);
  int* out = memoryManager::allocate<int>(N);

  unsigned* in_vals = memoryManager::allocate<unsigned>(N);
  unsigned* out_vals = memoryManager::allocate<unsigned>(N);

  std::iota(in      , in + N/2, 0);
  std::iota(in + N/2, in + N  , 0);
  std::shuffle(in      , in + N/2, std::mt19937{12345u});
  std::shuffle(in + N/2, in + N  , std::mt19937{67890u});

  std::fill(in_vals      , in_vals + N/2, 0);
  std::fill(in_vals + N/2, in_vals + N  , 1);
```

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

```
6 7 2 1 0 9 4 8 5 3 4 9 6 3 7 0 1 8 2 5
```

and the following sequence of (key, value) pairs in the `in` and `in_vals` arrays:

```
(6,0) (7,0) (2,0) (1,0) (0,0) (9,0) (4,0) (8,0) (5,0) (3,0)
(4,1) (9,1) (6,1) (3,1) (7,1) (0,1) (1,1) (8,1) (2,1) (5,1)
```

Unstable Sorts

A sequential unstable sort operation is performed by:

```
  RAJA::sort<RAJA::seq_exec>(RAJA::make_span(out, N));
```

Since no comparator is passed to the sort method, the default less operation is applied and the result generated in the out array is non-decreasing sort on the out array. The resulting out array contains the values:

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

We can be explicit about the operation used in the sort by passing the less operator to the sort method:

```
RAJA::sort<RAJA::seq_exec>(RAJA::make_span(out, N),
                           RAJA::operators::less<int>{});
```

The result in the out array is the same.

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

```
RAJA::sort<RAJA::omp_parallel_for_exec>(RAJA::make_span(out, N),
                                        RAJA::operators::less<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 sort on a GPU using CUDA, we would use a CUDA execution policy. This will be shown shortly.

Stable Sorts

A sequential stable sort (less) operation is performed by:

```
RAJA::stable_sort<RAJA::seq_exec>(RAJA::make_span(out, N),
                                  RAJA::operators::less<int>{});
```

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

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

Note that the stable sort result is the same as the unstable sort in this case because we are sorting integers. We will show an example of sorting pairs later where this is not the case.

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

```
RAJA::stable_sort<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(RAJA::make_span(out, N),
                                                    RAJA::operators::less<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.

Other Comparators

Using a different comparator allows sorting in a different order. Here is a sequential stable sort that uses the greater operator:

```
RAJA::stable_sort<RAJA::seq_exec>(RAJA::make_span(out, N),
                                  RAJA::operators::greater<int>{});
```

This generates the following sequence of values in non-increasing order in the output array:

```
9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1 0 0
```

Note that the only operators provided by RAJA that are valid to use in sort because they form a strict weak ordering of elements for arithmetic types are less and greater. Also note that the the cuda sort backend only supports RAJA's operators less and greater.

Sort Pairs

Sort *Pairs* operations generate the same results as the sort operations we have just described. However, an additional array of values is also permuted to match the sorted array so **two arrays are passed to sort pairs methods.**

Here is a sequential unstable sort pairs that uses the less operator:

```
  RAJA::sort_pairs<RAJA::seq_exec>(RAJA::make_span(out, N),
                                   RAJA::make_span(out_vals, N),
                                   RAJA::operators::less<int>{});
```

This generates the following sequence in the output array:

```
(0,0) (0,1) (1,0) (1,1) (2,0) (2,1) (3,0) (3,1) (4,0) (4,1)
(5,1) (5,0) (6,1) (6,0) (7,0) (7,1) (8,0) (8,1) (9,1) (9,0)
```

Note that some of the pairs with equivalent keys stayed in the same order they appeared in the unsorted arrays like `(8,0) (8,1)`, while others are reversed like `(9,1) (9,0)`.

Here is a sequential stable sort pairs that uses the greater operator:

```
  RAJA::stable_sort_pairs<RAJA::seq_exec>(RAJA::make_span(out, N),
                                          RAJA::make_span(out_vals, N),
                                          RAJA::operators::greater<int>{});
```

This generates the following sequence in the output array:

```
(9,0) (9,1) (8,0) (8,1) (7,0) (7,1) (6,0) (6,1) (5,0) (5,1)
(4,0) (4,1) (3,0) (3,1) (2,0) (2,1) (1,0) (1,1) (0,0) (0,1)
```

Note that all pairs with equivalent keys stayed in the same order that they appeared in the unsorted arrays.

As you may expect at this point, running an stable sort pairs operation using OpenMP is accomplished by:

```
  RAJA::stable_sort_pairs<RAJA::omp_parallel_for_exec>(RAJA::make_span(out, N),
                                                       RAJA::make_span(out_vals, N),
                                                       RAJA::operators::greater<int>{}
↪);
```

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

```
(9,0) (9,1) (8,0) (8,1) (7,0) (7,1) (6,0) (6,1) (5,0) (5,1)
(4,0) (4,1) (3,0) (3,1) (2,0) (2,1) (1,0) (1,1) (0,0) (0,1)
```

and the only difference is the execution policy template parameter.

Lastly, we show a parallel unstable sort pairs operation using CUDA:

```
  RAJA::sort_pairs<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(RAJA::make_span(out, N),
                                                     RAJA::make_span(out_vals, N),
                                                     RAJA::operators::greater<int>{});
```

Note: RAJA sorts for the HIP back-end are similar to those for CUDA.

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

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
- Specifying lambda arguments through statements

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 this:

```
  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 presented below, we 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);
```

We show the most basic RAJA view usage here – to simplify multi-dimensional array indexing. RAJA views can be used to abstract a variety of different data layouts and access patterns, including stride permutations, offsets, etc. For more information about RAJA views, see *View and Layout*.

We also use the following `RAJA::RangeSegment` 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);
```

Should I Use RAJA::forall For Nested Loops?

We begin by walking through some RAJA variants 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);
    }
    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 execution policy in the earlier case that used a `RAJA::forall` statement for the outer loop.

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 the `RAJA::kernel` interface, which was 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
        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, `RAJA::kernel` expresses 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` involves 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 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 only the policy and leaving the loop kernel code as is.

If we want to execute the row loop using OpenMP multithreaded 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 emphasize 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_POL6a =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::loop_exec,
      RAJA::statement::For<0, RAJA::loop_exec,
```

(continues on next page)

(continued from previous page)

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

RAJA::kernel_param<EXEC_POL6a>(
  RAJA::make_tuple(col_range, row_range, dot_range),

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

  // lambda 0
  [=] (double& dot) {
     dot = 0.0;
  },

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

  // lambda 2
  [=] (int col, int row, 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 the 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 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 operation (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 variables to be used in all three lambda expressions. However, observe that not all lambda expressions use all three index variables. They are declared, but left unnamed to prevent compiler warnings.

Alternatively, the lambda statements in the execution policy may be used to specify which arguments each lambda takes and in which order. For example:

```
// Alias for convenience
using RAJA::Segs;
using RAJA::Params;

using EXEC_POL6b =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::loop_exec,
      RAJA::statement::For<0, RAJA::loop_exec,
```

(continues on next page)

(continued from previous page)

```
        RAJA::statement::Lambda<0, Params<0>>,  // dot = 0.0
        RAJA::statement::For<2, RAJA::loop_exec,
          RAJA::statement::Lambda<1, Segs<0,1,2>, Params<0>> // dot += ...
        >,
        RAJA::statement::Lambda<2, Segs<0,1>, Params<0>>  // C(row, col) = dot
      >
    >
  >;

RAJA::kernel_param<EXEC_POL6b>(
  RAJA::make_tuple(col_range, row_range, dot_range),

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

  // lambda 0
  [=] (double& dot) {
     dot = 0.0;
  },

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

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

);
```

By using `RAJA::statement::Lambda` parameters in this way, the code potentially indicates more clearly which areguments are used. Of course, this makes the execution policy more verbose, but that is typically hidden away in a header file. Statements such as `RAJA::Segs`, and `RAJA::Params` identify the positions of the segments and params in the tuples to be used as arguments to the lambda expressions.

As we noted earlier, 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 in the file listed below to see that this is indeed the case. In the interest of simplicity, the remaining matrix multiplication examples do not use `RAJA::statement::Lambda` parameters to specify arguments to the lambda expressions.

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 into one OpenMP parallel region:

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

(continues on next page)

(continued from previous page)

```
        RAJA::statement::Lambda<2, RAJA::Segs<0, 1>, RAJA::Params<0>>   // 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 show how to use RAJA::statement::CudaKernel and RAJA::statement::HipKernel types to generate GPU kernels 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 the x dimension of each block:

```
  using EXEC_POL8 =
    RAJA::KernelPolicy<
      RAJA::statement::CudaKernel<
        RAJA::statement::For<1, RAJA::cuda_block_x_loop,    // row
          RAJA::statement::For<0, RAJA::cuda_thread_x_loop, // col
            RAJA::statement::Lambda<0, RAJA::Params<0>>,    // dot = 0.0
            RAJA::statement::For<2, RAJA::seq_exec,
                RAJA::statement::Lambda<1>                   // dot += ...
            >,
            RAJA::statement::Lambda<2, RAJA::Segs<0, 1>, RAJA::Params<0>>   // 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 HIP execution policy is similar:

```
  using EXEC_POL8 =
    RAJA::KernelPolicy<
      RAJA::statement::HipKernel<
        RAJA::statement::For<1, RAJA::hip_block_x_loop,    // row
          RAJA::statement::For<0, RAJA::hip_thread_x_loop, // col
            RAJA::statement::Lambda<0, RAJA::Params<0>>,  // dot = 0.0
            RAJA::statement::For<2, RAJA::seq_exec,
                RAJA::statement::Lambda<1>                  // dot += ...
            >,
            RAJA::statement::Lambda<2,
              RAJA::Segs<0,1>, RAJA::Params<0>>            // set C = ...
          >
        >
```

(continues on next page)

(continued from previous page)

```
      >
    >;
```

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_POL9a =
    RAJA::KernelPolicy<
      RAJA::statement::CudaKernel<
        RAJA::statement::Tile<1, RAJA::tile_fixed<CUDA_BLOCK_SIZE>,
                                 RAJA::cuda_block_y_loop,
          RAJA::statement::Tile<0, RAJA::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, RAJA::Params<0>>,     // dot = 0.0
                RAJA::statement::For<2, RAJA::seq_exec,
                    RAJA::statement::Lambda<1>                  // dot += ...
                >,
                RAJA::statement::Lambda<2, RAJA::Segs<0, 1>, RAJA::Params<0>>   //␣
↪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.

The analogous HIP policy is:

```
  using EXEC_POL9b =
    RAJA::KernelPolicy<
      RAJA::statement::HipKernel<
        RAJA::statement::Tile<1, RAJA::tile_fixed<HIP_BLOCK_SIZE>,
                                 RAJA::hip_block_y_loop,
          RAJA::statement::Tile<0, RAJA::tile_fixed<HIP_BLOCK_SIZE>,
                                   RAJA::hip_block_x_loop,
            RAJA::statement::For<1, RAJA::hip_thread_y_loop,     // row
              RAJA::statement::For<0, RAJA::hip_thread_x_loop,  // col
                RAJA::statement::Lambda<0, Params<0>>,           // dot = 0.0
                RAJA::statement::For<2, RAJA::seq_exec,
                  RAJA::statement::Lambda<1, Segs<0,1,2>, Params<0>> // dot += ...
                >,
                  RAJA::statement::Lambda<2, Segs<0,1>, Params<0>>   // set C = ...
              >
            >
          >
        >
      >
    >;
```

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, plus others that show a variety of `RAJA::kernel` execution policy types. 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>
                            >
                          >
                        >
                      >;

  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 of the three levels 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 kernel executes. 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'.

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; i < 2; ++i) {
      // print loop index triple...
    }
  }
}
```

The integer argument to each statement::For type is needed so that the levels in the loop nest can be reordered by changing the policy while the kernel remains the same. 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) {
```

(continues on next page)

(continued from previous page)

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

Finally, for completeness, 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>
                            >
                          >
                        >
                      >;
```

The 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.cpp` 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.,

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

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

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

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*.

The views for layout 1 are constructed as follows:

```
std::array<RAJA::idx_t, 3> perm1 {{0, 1, 2}};
auto layout1 =
    RAJA::make_permuted_layout( {{N, N_r, N_c}}, perm1 );

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 ($N_c \times N_r$).

The views for layout 2 are 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

Complete working examples that run the batched matrix-multiplication computation for both layouts and various RAJA execution policies is located in the file `RAJA/examples/tut_batched-matrix-multiply.cpp`.

It compares the execution run times of the two layouts described above using four RAJA back-ends (Sequential, OpenMP, CUDA, and HIP). The OpenMP version for layout 1 looks like this:

```
    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);

        });
```

The only differences between the lambda loop body for layout 1 and layout 2 cases are the names of the views. To make the algorithm code identical for all cases, we would use type aliases for the view and layout types in a header file similarly to how we would abstract the execution policy out of the algorithm.

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+1, N_c+1}});

RAJA::View<int, RAJA::OffsetLayout<DIM>> inputView(input, layout);
RAJA::View<int, RAJA::OffsetLayout<DIM>> outputView(output, 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) {

                                 outputView(row, col) =
                                     inputView(row, col)
                                     + inputView(row - 1, col)
                                     + inputView(row + 1, col)
                                     + inputView(row, col - 1)
                                     + inputView(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 information about using the RAJA::kernel interface, please see *Complex Loops (RAJA::kernel)*.

Tiled Matrix Transpose

Key RAJA features shown in this example are:

- RAJA::kernel usage with multiple lambdas
- RAJA::statement::Tile 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 storing the tile in another array. The algorithm is expressed as a collection of outer and inner loops. Iterations of the inner loop will transpose each tile, while outer loops iterate over the tiles.

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. First, we define matrix dimensions:

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

const int TILE_DIM = 16;

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

Then, we wrap the matrix data pointers in `RAJA::View` objects to simplify the multi-dimensional indexing:

```
RAJA::View<int, RAJA::Layout<DIM>> Aview(A, N_r, N_c);
RAJA::View<int, RAJA::Layout<DIM>> Atview(At, N_c, N_r);
```

Then, the non-RAJA C++ implementation looks like this:

```
//
// (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 need to 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 Variants

For `RAJA::kernel` variants, we use `RAJA::statement::Tile` types for the outer loop tiling and `RAJA::tile_fixed` types to indicate the tile dimensions. The complete sequential RAJA variant is:

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

(continues on next page)

(continued from previous page)

```
      >
    >
  >;

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

The `RAJA::statement::Tile` types compute the number of tiles needed to iterate over all matrix entries in each dimension and generate iteration index bounds for each tile, which are used to generate loops for the inner `RAJA::statement::For` types. Thus, the bounds checking logic in the non-RAJA variant is not needed. Note that the integer template parameters to these statement types refer to the entries in the iteration space tuple passed to the `RAJA::kernel` method.

The file `RAJA/examples/tut_tiled-matrix-transpose.cpp` contains the complete working example code for the examples described in this section, including OpenMP, CUDA, and HIP 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 the discussion in *Tiled Matrix Transpose*, where only loop tiling is considered. Here, we combine loop tiling with `RAJA::LocalArray` objects which enable us to store data for each tile in CPU stack-allocated arrays or 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 include:

- `RAJA::kernel_param` method with multiple lambda expressions
- `RAJA::statement::Tile` type
- `RAJA::statement::ForICount` type
- `RAJA::LocalArray`
- Specifying lambda arguments through statements

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 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. As in the *Tiled Matrix Transpose* example, we start by defining the number of rows and columns in the matrices, the tile dimensions, and the number of tiles.

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

const int TILE_DIM = 16;

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

We also use RAJA View objects to simplify the multi-dimensional indexing as in the *Tiled Matrix Transpose* example.

```
RAJA::View<int, RAJA::Layout<DIM>> Aview(A, N_r, N_c);
RAJA::View<int, RAJA::Layout<DIM>> Atview(At, N_c, N_r);
```

The complete sequential C++ implementation of the tiled transpose operation using a stack-allocated local array for the tiles 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 of Tiled Loops with Local Array

RAJA provides mechanisms to tile loops and use *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 and create an object to represent it:

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

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. Next, we compare two RAJA implementations of matrix transpose with RAJA.

The complete RAJA sequential CPU variant with kernel execution policy and kernel is:

```
using SEQ_EXEC_POL_I =
  RAJA::KernelPolicy<
    RAJA::statement::Tile<1, RAJA::tile_fixed<TILE_DIM>, RAJA::loop_exec,
      RAJA::statement::Tile<0, RAJA::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>
          >
        >

        >
      >
    >
```

(continues on next page)

(continued from previous page)

```
  >;

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

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

  [=](int col, int row, int tx, int ty, TILE_MEM &Tile_Array) {
    Tile_Array(ty, tx) = Aview(row, col);
  },

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

});
```

The RAJA::statement::Tile types in 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::tile_fixed types) and loop execution policies. Next, the RAJA::statement::InitLocalMem type initializes the local stack array based on the memory policy type (here, we use RAJA::cpu_tile_mem for a CPU stack-allocated array). 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. The first two entries in the parameter tuple indicate storage for the local tile indices which can be used in multiple lambdas in the kernel. 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.

Note that the loops over tiles use RAJA::statement::ForICount types rather than RAJA::statement::For types that we have seen in other nested loop examples. The RAJA::statement::ForICount type generates local tile indices that are passed to lambda loop body expressions. As the 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 have 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 ranges for the 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.

RAJA::kernel Version of Tiled Loops with Local Array Specifying Lambda Arguments

The second RAJA variant works the same as the one above. The main differences between the two variants is due to the fact that in this second one, we use `RAJA::statement::Lambda` types to indicate which arguments each lambda takes and in which order. Here is the complete version including execution policy and kernel:

```
using SEQ_EXEC_POL_II =
  RAJA::KernelPolicy<
    RAJA::statement::Tile<1, RAJA::tile_fixed<TILE_DIM>, RAJA::loop_exec,
      RAJA::statement::Tile<0, RAJA::tile_fixed<TILE_DIM>, RAJA::loop_exec,

        RAJA::statement::InitLocalMem<RAJA::cpu_tile_mem, RAJA::ParamList<0>,

        RAJA::statement::For<1, RAJA::loop_exec,
          RAJA::statement::For<0, RAJA::loop_exec,
            RAJA::statement::Lambda<0, Segs<0>, Segs<1>, Offsets<0>, Offsets<1>, 
↪Params<0> >
          >
        >,

        RAJA::statement::For<0, RAJA::loop_exec,
          RAJA::statement::For<1, RAJA::loop_exec,
            RAJA::statement::Lambda<1, Segs<0, 1>, Offsets<0, 1>, Params<0> >
          >
        >

        >
      >
    >
  >;

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

  RAJA::make_tuple(Tile_Array),

  [=](int col, int row, int tx, int ty, TILE_MEM &Tile_Array) {
      Tile_Array(ty, tx) = Aview(row, col);
  },

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

});
```

Here, the two `RAJA::statement::Lambda` types in the execution policy show two different ways to specify the segments (`RAJA::Segs`) associated with the matrix column and row indices. That is, we can use a `Segs` statement for each argument, or include multiple segment ids in one statement.

Note that we are using `RAJA::statement::For` types for the inner tile loops instead of *RAJA::statement::ForICount‘* types used in the first variant. As a consequence of specifying lambda arguments, there are two main differences. The local tile indices are properly computed and passed to the lambda expressions as a result of the `RAJA::Offsets` types that appear in the lambda statement types. The `RAJA::statement::Lambda` type for each lambda shows the two ways to specify the local tile index args; we can use an `Offsets` statement for each argument, or include multiple segment ids in one statement. Lastly, there is only one entry in the parameter tuple in this case, the local tile array. The placeholders are not needed.

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, CUDA, and HIP variants.

Halo Exchange (Workgroup Constructs)

Key RAJA features shown in this example:

- `RAJA::WorkPool` workgroup construct
- `RAJA::WorkGroup` workgroup construct
- `RAJA::WorkSite` workgroup construct
- `RAJA::RangeSegment` iteration space construct
- RAJA workgroup policies

In this example, we show how to use the RAJA workgroup constructs to implement buffer packing and unpacking for data halo exchange on a computational grid, a common MPI communication operation. This may not provide a performance gain on a CPU system, but it can significantly speedup halo exchange on a GPU system compared to using `RAJA::forall` to run individual packing/unpacking kernels.

Note: Using an abstraction layer over RAJA can make it easy to switch between using individual `RAJA::forall` loops or the RAJA workgroup constructs to implement halo exchange packing and unpacking at compile time or run time.

We start by setting the parameters for the halo exchange by using default values or values provided via command line input. These parameters determine the size of the grid, the width of the halo, the number of grid variables and the number of cycles.

```
//
// Define grid dimensions
// Define halo width
// Define number of grid variables
// Define number of cycles
//
const int grid_dims[3] = { (argc != 7) ? 100 : std::atoi(argv[1]),
                           (argc != 7) ? 100 : std::atoi(argv[2]),
                           (argc != 7) ? 100 : std::atoi(argv[3]) };
const int halo_width =     (argc != 7) ?   1 : std::atoi(argv[4]);
const int num_vars   =     (argc != 7) ?   3 : std::atoi(argv[5]);
const int num_cycles =     (argc != 7) ?   3 : std::atoi(argv[6]);
```

Next, we allocate the variables array (the memory manager in the example uses CUDA Unified Memory if CUDA is enabled). These grid variables are reset each cycle to allow checking the results of the packing and unpacking.

```
//
// Allocate grid variables and reference grid variables used to check
// correctness.
//
std::vector<double*> vars    (num_vars, nullptr);
std::vector<double*> vars_ref(num_vars, nullptr);

for (int v = 0; v < num_vars; ++v) {
  vars[v]     = memoryManager::allocate<double>(var_size);
  vars_ref[v] = memoryManager::allocate<double>(var_size);
}
```

We also allocate and initialize index lists of the grid elements to pack and unpack:

All the code examples presented below copy the data packed from the grid interior:

0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

into the adjacent halo cells:

1	1	2	3	3
1	1	2	3	3
4	4	5	6	6
7	7	8	9	9
7	7	8	9	9

Packing and Unpacking (Basic Loop Execution)

A sequential non-RAJA example of packing:

```
      for (int l = 0; l < num_neighbors; ++l) {

        double* buffer = buffers[l];
        int* list = pack_index_lists[l];
        int  len  = pack_index_list_lengths[l];

        // pack
        for (int v = 0; v < num_vars; ++v) {

          double* var = vars[v];

          for (int i = 0; i < len; i++) {
            buffer[i] = var[list[i]];
          }

          buffer += len;
        }

        // send single message
      }
```

and unpacking:

```
      for (int l = 0; l < num_neighbors; ++l) {

        // recv single message

        double* buffer = buffers[l];
        int* list = unpack_index_lists[l];
        int  len  = unpack_index_list_lengths[l];

        // unpack
```

(continues on next page)

(continued from previous page)

```
      for (int v = 0; v < num_vars; ++v) {

        double* var = vars[v];

        for (int i = 0; i < len; i++) {
          var[list[i]] = buffer[i];
        }

        buffer += len;
      }
    }
```

RAJA Variants using forall

A sequential RAJA example uses this execution policy type:

```
  using forall_policy = RAJA::loop_exec;
```

to pack the grid variable data into a buffer:

```
    for (int l = 0; l < num_neighbors; ++l) {

      double* buffer = buffers[l];
      int* list = pack_index_lists[l];
      int  len  = pack_index_list_lengths[l];

      // pack
      for (int v = 0; v < num_vars; ++v) {

        double* var = vars[v];

        RAJA::forall<forall_policy>(range_segment(0, len), [=] (int i) {
          buffer[i] = var[list[i]];
        });

        buffer += len;
      }

      // send single message
    }
```

and unpack the buffer data into the grid variable array:

```
    for (int l = 0; l < num_neighbors; ++l) {

      // recv single message

      double* buffer = buffers[l];
      int* list = unpack_index_lists[l];
      int  len  = unpack_index_list_lengths[l];

      // unpack
      for (int v = 0; v < num_vars; ++v) {

        double* var = vars[v];
```

(continues on next page)

(continued from previous page)

```
      RAJA::forall<forall_policy>(range_segment(0, len), [=] (int i) {
        var[list[i]] = buffer[i];
      });

      buffer += len;
    }
  }
```

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

```
using forall_policy = RAJA::omp_parallel_for_exec;
```

Similarly, to run the loops in parallel on a CUDA GPU, we would use this policy:

```
using forall_policy = RAJA::cuda_exec_async<CUDA_BLOCK_SIZE>;
```

RAJA Variants using workgroup constructs

Using the workgroup constructs in the example requires defining a few more policies and types:

```
using forall_policy = RAJA::loop_exec;

using workgroup_policy = RAJA::WorkGroupPolicy <
                             RAJA::loop_work,
                             RAJA::ordered,
                             RAJA::ragged_array_of_objects >;

using workpool = RAJA::WorkPool< workgroup_policy,
                                 int,
                                 RAJA::xargs<>,
                                 memory_manager_allocator<char> >;

using workgroup = RAJA::WorkGroup< workgroup_policy,
                                   int,
                                   RAJA::xargs<>,
                                   memory_manager_allocator<char> >;

using worksite = RAJA::WorkSite< workgroup_policy,
                                 int,
                                 RAJA::xargs<>,
                                 memory_manager_allocator<char> >;
```

which are used in a slightly rearranged version of packing. See how the comment indicating where a message could be sent has been moved down after the call to run on the workgroup:

```
  for (int l = 0; l < num_neighbors; ++l) {

    double* buffer = buffers[l];
    int* list = pack_index_lists[l];
    int  len  = pack_index_list_lengths[l];

    // pack
    for (int v = 0; v < num_vars; ++v) {
```

(continues on next page)

(continued from previous page)

```
        double* var = vars[v];

        pool_pack.enqueue(range_segment(0, len), [=] (int i) {
          buffer[i] = var[list[i]];
        });

        buffer += len;
      }
    }

    workgroup group_pack = pool_pack.instantiate();

    worksite site_pack = group_pack.run();

    // send all messages
```

Similarly, in the unpacking we wait to receive all of the messages before unpacking the data:

```
    // recv all messages

    for (int l = 0; l < num_neighbors; ++l) {

      double* buffer = buffers[l];
      int* list = unpack_index_lists[l];
      int  len  = unpack_index_list_lengths[l];

      // unpack
      for (int v = 0; v < num_vars; ++v) {

        double* var = vars[v];

        pool_unpack.enqueue(range_segment(0, len), [=] (int i) {
          var[list[i]] = buffer[i];
        });

        buffer += len;
      }
    }

    workgroup group_unpack = pool_unpack.instantiate();

    worksite site_unpack = group_unpack.run();
```

This reorganization has the downside of not overlapping the message sends with packing and the message receives with unpacking.

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

```
  using forall_policy = RAJA::omp_parallel_for_exec;

  using workgroup_policy = RAJA::WorkGroupPolicy <
                               RAJA::omp_work,
                               RAJA::ordered,
                               RAJA::ragged_array_of_objects >;

  using workpool = RAJA::WorkPool< workgroup_policy,
```

(continues on next page)

(continued from previous page)

```
                                     int,
                                     RAJA::xargs<>,
                                     memory_manager_allocator<char> >;

    using workgroup = RAJA::WorkGroup< workgroup_policy,
                                       int,
                                       RAJA::xargs<>,
                                       memory_manager_allocator<char> >;

    using worksite = RAJA::WorkSite< workgroup_policy,
                                     int,
                                     RAJA::xargs<>,
                                     memory_manager_allocator<char> >;
```

Similarly, to run the loops in parallel on a CUDA GPU use these policies and types, taking note of the unordered work ordering policy that allows the enqueued loops to all be run using a single CUDA kernel:

```
    using forall_policy = RAJA::cuda_exec_async<CUDA_BLOCK_SIZE>;

    using workgroup_policy = RAJA::WorkGroupPolicy <
                                 RAJA::cuda_work_async<CUDA_WORKGROUP_BLOCK_SIZE>,
                                 RAJA::unordered_cuda_loop_y_block_iter_x_threadblock_
↪average,
                                 RAJA::constant_stride_array_of_objects >;

    using workpool = RAJA::WorkPool< workgroup_policy,
                                     int,
                                     RAJA::xargs<>,
                                     pinned_allocator<char> >;

    using workgroup = RAJA::WorkGroup< workgroup_policy,
                                       int,
                                       RAJA::xargs<>,
                                       pinned_allocator<char> >;

    using worksite = RAJA::WorkSite< workgroup_policy,
                                     int,
                                     RAJA::xargs<>,
                                     pinned_allocator<char> >;
```

The packing is the same as the previous workgroup packing examples with the exception of added synchronization after calling run and before sending the messages. The previous CUDA example used forall to launch `num_neighbors * num_vars` CUDA kernels and performed `num_neighbors` synchronizations to send each message in turn. Here, the reorganization to pack all messages before sending lets us use an unordered CUDA work ordering policy in the workgroup constructs that reduces the number of CUDA kernel launches to one. It also allows us to synchronize once before sending all of the messages:

```
      for (int l = 0; l < num_neighbors; ++l) {

        double* buffer = buffers[l];
        int* list = pack_index_lists[l];
        int  len  = pack_index_list_lengths[l];

        // pack
        for (int v = 0; v < num_vars; ++v) {
```

(continues on next page)

(continued from previous page)

```
    double* var = vars[v];

    pool_pack.enqueue(range_segment(0, len), [=] RAJA_DEVICE (int i) {
      buffer[i] = var[list[i]];
    });

    buffer += len;
  }
}

workgroup group_pack = pool_pack.instantiate();

worksite site_pack = group_pack.run();

cudaErrchk(cudaDeviceSynchronize());

// send all messages
```

After waiting to receive all of the messages we use workgroup constructs using a CUDA unordered work ordering policy to unpack all of the messages using a single kernel launch:

```
// recv all messages

for (int l = 0; l < num_neighbors; ++l) {

  double* buffer = buffers[l];
  int* list = unpack_index_lists[l];
  int  len  = unpack_index_list_lengths[l];

  // unpack
  for (int v = 0; v < num_vars; ++v) {

    double* var = vars[v];

    pool_unpack.enqueue(range_segment(0, len), [=] RAJA_DEVICE (int i) {
      var[list[i]] = buffer[i];
    });

    buffer += len;
  }
}

workgroup group_unpack = pool_unpack.instantiate();

worksite site_unpack = group_unpack.run();

cudaErrchk(cudaDeviceSynchronize());
```

Note that the synchronization after unpacking is done to ensure that `group_unpack` and `site_unpack` survive until the unpacking loop has finished executing.

The file `RAJA/examples/tut_halo-exchange.cpp` contains a complete working example code, with OpenMP, CUDA, and HIP variants.

Team based Loops: Nested loops with a thread/team model

The examples in this section illustrate how to use RAJA::expt::launch to create an run-time selectable execution space for expressing algorithms in terms of threads and teams.

Team based loops (RAJA Teams)

Key RAJA features shown in the following examples:

- RAJA::expt::launch method to create a run-time selectable host/device execution space.
- RAJA::expt::loop methods to express algorithms in terms of nested for loops.

In this example, we introduce the RAJA Teams framework and discuss hierarchical loop-based parallelism. Development with RAJA Teams occurs inside an execution space. The execution space is launched using the RAJA::expt::launch method:

```
RAJA::expt::launch<launch_policy>(RAJA::expt::ExecPlace ,
RAJA::expt::Grid(RAJA::expt::Teams(Nteams,Nteams),
                      RAJA::expt::Threads(Nthreads,Nthreads)),
[=] RAJA_HOST_DEVICE (RAJA::expt::LaunchContext ctx) {

  /* Express code here */

});
```

The RAJA::expt::launch method is templated on both a host and a device launch policy. As an example, the following constructs an execution space for a sequential and CUDA kernel:

```
using launch_policy = RAJA::expt::LaunchPolicy
  <RAJA::expt::seq_launch_t, RAJA::expt::cuda_launch_t<false>>;
```

Kernel execution on either the host or device is driven by the first argument of the method which takes a RAJA::expt::ExecPlace enum type, either HOST or DEVICE. Similar to thread, and block programming models, RAJA Teams carries out computation in a predefined compute grid made up of threads which are then grouped into teams. The execution space is then enclosed by a host/device lambda which takes a RAJA::expt::LaunchContext object. The RAJA::expt::LaunchContext may then be used to control the flow within the kernel, for example creating thread-team synchronization points.

Inside the execution space the RAJA::expt::loop methods enable developers to express their code in terms of nested loops. The manner in which the loops are executed depends on the template. Following the CUDA/HIP programming models we follow a hierarchical structure in which outer loops are executed by thread-teams and inner loops are executed by a thread in a team.

```
   RAJA::expt::loop<teams_y>(ctx, RAJA::RangeSegment(0, Nteams), [&] (int by) {
     RAJA::expt::loop<teams_x>(ctx, RAJA::RangeSegment(0, Nteams), [&] (int bx) {

       RAJA::expt::loop<threads_y>(ctx, RAJA::RangeSegment(0, Nthreads), [&] (int ty)
↪{
         RAJA::expt::loop<threads_x>(ctx, RAJA::RangeSegment(0, Nthreads), [&] (int␣
↪tx) {

             printf("RAJA Teams: threadId_x %d threadId_y %d teamId_x %d teamId_y %d␣
↪\n",
                    tx, ty, bx, by);

           });
```

(continues on next page)

(continued from previous page)

```
         });

       });
     });
```

The mapping between the thread and teams to programming model depends on how they are defined. For example, we may define host and device mapping strategies as the following:

```
using teams_x = RAJA::expt::LoopPolicy<RAJA::loop_exec,
                                       RAJA::cuda_block_x_direct>;
using thread_x = RAJA::expt::LoopPolicy<RAJA::loop_exec,
                                        RAJA::cuda_block_x_direct>;
```

In the example above the `RAJA::expt::LoopPolicy` struct holds both the host and device loop mapping strategies. On the host, both the team/thread strategies expand out to standard C-style loops for execution:

```
    for(int by=0; by<Nteams; ++by) {
      for(int bx=0; bx<Nteams; ++bx) {

        for(int ty=0; ty<Nthreads; ++ty) {
          for(int tx=0; tx<Nthreads; ++tx) {

            printf("c-iter: iter_tx %d iter_ty %d iter_bx %d iter_by %d \n",
                   tx, ty, bx, by);
          }
        }

      }
    }
```

On the device the `teams_x/y` policies will map loop iterations directly to CUDA thread blocks, while the `thread_x/y` policies will map loop iterations directly to threads in a CUDA block. The CUDA equivalent is illustrated below:

```
  {int by = blockIdx.y;
    {int bx = blockIdx.x;

      {int ty = threadIdx.y;
        {int tx = blockIdx.x;

          printf("device-iter: threadIdx_tx %d threadIdx_ty %d block_bx %d block_by
↪%d \n",
                 tx, ty, bx, by);

        }
      }

    }
  }
```

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

Naming kernels with NVTX/ROCTX tools

Key RAJA feature shown in the following example:

- Naming kernels using the `Grid` object in `RAJA::ext::Launch` methods.

In this example we illustrate kernel naming capabilities within the RAJA Teams framework for use with NVTX or ROCTX region naming capabilities.

Recalling the `RAJA::expt::launch` API, naming a kernel is done using the third argument of the `Resources` constructor as illustrated below:

```
RAJA::expt::launch<launch_policy>(RAJA::expt::ExecPlace ,
RAJA::expt::Grid(RAJA::expt::Teams(Nteams,Nteams),
                      RAJA::expt::Threads(Nthreads,Nthreads)
                      "myKernel"),
[=] RAJA_HOST_DEVICE (RAJA::expt::LaunchContext ctx) {

  /* Express code here */

});
```

The kernel name is used to create NVTX (NVIDIA) or ROCTX (AMD) ranges enabling developers to identify kernels using NVIDIA Nsight and NVIDIA Nvprof profiling tools or ROCm profiling tools when using ROCTX. As an illustration, using Nvprof kernels are identified as ranges of GPU activity through the user specified name:

```
==73220== NVTX result:
==73220==   Thread "<unnamed>" (id = 290832)
==73220==     Domain "<unnamed>"
==73220==       Range "myKernel"
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
          Range:  100.00%  32.868us         1  32.868us  32.868us  32.868us ␣
↪myKernel
   GPU activities:  100.00%  2.0307ms         1  2.0307ms  2.0307ms  2.0307ms  _
↪ZN4RAJA4expt17launch_global_fcnIZ4mainEUlNS0_13LaunchContextEE_EEvS2_T_
        API calls:  100.00%  27.030us         1  27.030us  27.030us  27.030us ␣
↪cudaLaunchKernel
```

In a similar fashion ROCm tools can be used to generate traces of the profile and the resulting json file can be viewed using tools such as perfetto.

As future work we plan to add support to other profiling tools; API changes may occur based on user feedback and integration with other tools. Enabling NVTX profiling with RAJA Teams requires RAJA to be configured with RAJA_ENABLE_NV_TOOLS_EXT=ON. or RAJA_ENABLE_ROCTX=ON for ROCTX profiling on AMD platforms platforms.

The file RAJA/examples/teams_reductions.cpp contains a complete working example code.

6.1.5 Using RAJA in Your Application

Using RAJA in an application requires two things: ensuring the RAJA header files are visible, and linking against the RAJA library. We maintain a RAJA Template Project that shows how to use RAJA in a project that uses CMake or make, either as a Git submodule or as an externally installed library that you link your application against.

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 a `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 PUBLIC RAJA)
```

6.1.6 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 RAJA configuration options that are most useful for users to know about and their defaults.

RAJA Option Types

Nearly all Cmake options used in RAJA contain the prefix `RAJA_` to give users flexibility to enable/disable individual compilation features for RAJA, specifically. RAJA contains two types of options, those that exist in RAJA only and those that are similar to standard CMake options or options provided by BLT; i.e., *dependent options* in CMake terminology. RAJA dependent option names are the same as the associated CMake and BLT option names, but with the `RAJA_` prefix added.

Note: RAJA uses a mix of RAJA-only options and CMake-dependent options that can be controlled with CMake or BLT variants.

- Dependent options are typically used for *disabling* features. For example, providing the option `-DRAJA_ENABLE_TESTS=Off` to CMake will disable compilation of RAJA tests, even if the option `-DENABLE_TESTS=On` is also provided.
- We recommend using the option names without the `RAJA_` prefix, when available, to enable features at compile time to avoid potential undesired behavior. For example, passing the option `-DRAJA_ENABLE_CUDA=On` to CMake will not enable CUDA because `ENABLE_CUDA` is off by default. So to enable CUDA, you need to pass the `-DENABLE_CUDA=On` option to Cmake.

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 \
```

(continues on next page)

(continued from previous page)

```
  -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`. With CMake, compiler flags for each of these build types are applied automatically and so you do not have to specify them. However, if you want to apply other compiler flags, you will need to do that using appropriate CMake variables.

All RAJA options are set like regular CMake variables. RAJA settings for default options, compilers, flags for optimization, etc. can be found in files in the `RAJA/cmake` directory and top-level `CMakeLists.txt` file. Configuration variables can be set by passing arguments to CMake on the command line when CMake is called, or by setting options in a CMake *cache file* and passing that file to CMake using the CMake `-C` options. For example, to enable RAJA OpenMP functionality, pass the following argument to CMake:

```
-DENABLE_OPENMP=On
```

The RAJA repository contains a collection of CMake cache files (we call them *host-config* files) that may be used as a guide for users trying to set their own options. See configopt-raja-hostconfig-label.

Next, we summarize RAJA options and their defaults.

Available RAJA 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 so all options propagate consistently through the build process for all of the code.

Note: The following discussion indicates which options exist in RAJA only and those which are dependent options. **Dependent options appear in the discussion with parentheses around the 'RAJA_' prefix to indicate that the option name without the prefix can be used and is often preferred.**

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

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

CMake variables can be used to control whether RAJA tests, examples, tutorial exercises, etc. are built when RAJA is compiled.

Variable	Default
(**RAJA_**)ENABLE_TESTS	On
(**RAJA_**)ENABLE_EXAMPLES	On
(**RAJA_**)ENABLE_BENCHMARKS	Off
(**RAJA_**)ENABLE_COVERAGE	Off (supported for GNU compilers only)
RAJA_ENABLE_EXERCISES	On
RAJA_ENABLE_REPRODUCERS	Off

RAJA can also be configured to build with compiler warnings reported as errors, which may be useful to make sure your application builds cleanly:

Variable	Default
(RAJA_)ENABLE_WARNINGS_AS_ERRORS	Off

RAJA Views/Layouts may be configured to check for out of bounds indexing at runtime:

Variable	Default
RAJA_ENABLE_BOUNDS_CHECK	Off

Note that RAJA bounds checking is a runtime check and will add considerable execution time overhead. Thus, this feature should only be used for correctness checking and should be disabled for production builds.

- **Programming model back-end support**

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

Variable	Default
(RAJA_)ENABLE_OPENMP	On
(RAJA_)ENABLE_CUDA	Off
(RAJA_)ENABLE_HIP	Off
RAJA_ENABLE_TARGET_OPENMP	Off (when on, (RAJA_)ENABLE_OPENMP must also be on!)
RAJA_ENABLE_TBB	Off
RAJA_ENABLE_SYCL	Off

Other programming model specific compilation options are also available:

Variable	Default
(RAJA_)ENABLE_CLANG_CUDA	Off (if on, (RAJA_)ENABLE_CUDA must be on too!)
RAJA_ENABLE_EXTERNAL_CUB	Off
RAJA_ENABLE_NV_TOOLS_EXT	Off
CUDA_ARCH	sm_35 (based on hardware support)
RAJA_ENABLE_EXTERNAL_ROCPRIM	Off
RAJA_ENABLE_ROCTX	Off
RAJA_ENABLE_HIP_INDIRECT_FUNCTION_CALL	Off (enables device function pointers in HIP back-end)

Turning the `(RAJA_)ENABLE_CLANG_CUDA` variable on will build CUDA code with the native support in the Clang compiler.

The `RAJA_ENABLE_EXTERNAL_CUB` variable is used to enable use of an external install of the NVIDIA CUB support library. When Off, the CUB library included in the CUDA toolkit will still be used, if available. Starting with CUDA 11, CUB is installed as part of the CUDA toolkit and the NVIDIA Thrust library requires that install of CUB. We recommended projects use the CUB included with the CUDA toolkit for compatibility with Thrust and applications using Thrust. Users should take note of the CUB install used by RAJA to ensure they use the same include directories when configuring their applications.

The `RAJA_ENABLE_EXTERNAL_ROCPRIM` variable is used to enable use of an external install of the AMD rocPRIM support library. When Off, the rocPRIM library included in the ROCm install will be used, when available. We recommend projects use the rocPRIM included with the ROCm install when available. Users should take note of the rocPRIM install used by RAJA to ensure they use the same include directories when configuring their applications.

Note: See *Getting Started With RAJA* for more information about setting other options for RAJA back-ends.

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

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

Note: RAJA data types in this section are provided as a convenience to users if they wish to use them. They are not used within RAJA implementation code directly.

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

Variable	Default
RAJA_USE_DOUBLE	On (type is double)
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 an alias to `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 a parameter to define platform-specific constant data alignment. The variable that control this is:

Variable	Default
RAJA_DATA_ALIGN	64

This variable is used to specify data alignment used in intrinsics and typedefs in 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.

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
RAJA_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.)
RAJA_ENABLE_RUNTIME_PLUGINS	Enable support for dynamically loaded RAJA plugins.
RAJA_ENABLE_DESUL_ATOMICS	Replace RAJA atomic implementations with desul variants at compile-time.

Setting RAJA Back-End Features

Various *ENABLE_** options are listed above for enabling RAJA back-ends, such as OpenMP and CUDA. To access compiler and hardware optimization features, it may be necessary to pass additional options to CMake. Please see *Getting Started With RAJA* for more information.

6.2 Doxygen

6.3 RAJA Developer Guide

The RAJA Developer Guide is a work-in-progress. . . .

This guide documents key software development processes used by the RAJA project so that they are understood and uniformly applied by contributors.

6.3.1 Contributing to RAJA

Since RAJA is a collaborative open source software project, we embrace contributions from anyone who wants to add features or improve its existing capabilities. This section describes basic processes to follow for individuals outside of the core RAJA team to contribute new features or bugfixes to RAJA. It assumes you are familiar with Git, which we use for source code version control, and GitHub, which is where our project is hosted.

This section describes development processes, such as:

- Making a fork of the RAJA repository
- Creating a branch for development
- Creating a pull request (PR)
- Tests that your PR must pass before it can be merged into RAJA

Forking RAJA

If you are not a member of the LLNL organization on GitHub and of the core RAJA team of developers, then you do not have permission to create a branch in the RAJA repository. This is due to the policy adopted by the LLNL organization on GitHub in which the RAJA project resides. Fortunately, you may still contribute to RAJA by forking the RAJA repo. Forking creates a copy of the RAJA repository that you own. You can push code changes on that copy to GitHub and create a pull request in the RAJA project.

Note: A contributor who is not a member of the LLNL GitHub organization and the core team of RAJA developers cannot create a branch in the RAJA repo. However, anyone can create a fork of the RAJA project and create a pull request in the RAJA project.

Developing RAJA Code

New features, bugfixes, and other changes are developed on a **feature branch.** Each such branch should be based on the RAJA `develop` branch. For more information on the branch development model used in RAJA, please see *RAJA Branch Development*. When you want to make a contribution, first ensure you have an up-to-date copy of the `develop` branch locally:

```
$ git checkout develop
$ git pull origin develop
```

Developing a Feature

Assuming you are on the develop branch in your local copy of the RAJA repo, and the branch is up-to-date, the first step toward developing a RAJA feature is to create a new branch on which to perform your development. For example:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to modify your branch by committing changes with reasonably-sized work portions (i.e., *atomic commits*), and add tests that will exercise your new code. If you are creating new functionality, please add documentation to the appropriate section of the RAJA User Guide. The source files for the RAJA documentation are maintained in the `RAJA/docs` directory.

After your new code is complete, you've tested it, and developed appropriate documentation, you can push your branch to GitHub and create a PR in the RAJA project. It will be reviewed by members of the RAJA team, who will provide comments, suggestions, etc. After it is approved and all CI checks pass, your contribution will be merged into the RAJA repository.

Important: When creating a branch that you intend to be merged into the RAJA repo, please give it a succinct name that clearly describes the contribution. For example, **feature/<name-of-feature>** for a new feature, **bugfix/<fixed-issue>** for a bugfix, etc.

Developing a Bug Fix

Contributing a bugfix follows the same process as described above. Be sure to indicate in the name of your branch that it is for a bugfix; for example:

```
$ git checkout -b bugfix/<fixed-issue>
```

We recommend that you add a test that reproduces the issue you have found and demonstrates that the issue is resolved. To verify that you have done this properly, build the code for your branch and then run `make test` to ensure that your new test passes.

When you are done, push your branch to GitHub, then create a PR in the RAJA project.

Creating a Pull Request

You can create a pull request (PR) here. GitHub has a good PR guide on PR basics if you want more information. Ensure that the base branch for your PR is the `develop` branch of RAJA.

When you create a RAJA PR, you must enter a description of the contents of the PR. We have a *PR template* for this purpose for you to fill in. Be sure to add a descriptive title explaining the bug you fixed or the feature you have added and any other relevant details that will assist the RAJA team in reviewing your contribution.

When a PR is created in RAJA, it will be run through our automated testing processes and be reviewed by RAJA team members. When the PR passes all tests and it is approved, a member of the RAJA team will merge it.

Note: Before a PR can be merged into RAJA, all CI checks must pass and the PR must be approved by a member of the core team.

Tests

RAJA uses multiple continuous integration (CI) tools to test every pull request. See *Continuous Integration (CI) Testing* for more information.

All RAJA tests are in the `RAJA/test` directory and are split into *unit tests* and *functional tests*. Unit tests are intended to test basic interfaces and features of individual classes, methods, etc. Functional tests are used to test combinations of RAJA features. We have organized our tests to make it easy to see what is being tested and easy to add new tests. For example, tests for each programming model back-end are exercised using the same common, parameterized test code to ensure back-end support is consistent.

Important: Please follow the sub-directory structure and code implementation pattern for existing tests in the `RAJA/test` directory when adding or modifying tests.

Testing Pull Requests from Branches in Forked Repositories

Due to LLNL security policies and RAJA project policies, only a PR created by someone on the RAJA core development team will be run automatically through all RAJA CI tools. In particular, a PR made from branch on a forked repository will not trigger Gitlab CI checks. Gitlab CI on internal LLNL platforms will only be run on PRs that are made from branches in the GitHub RAJA repository. This may change in the future to make it easier to work with PRs from contributors that are not members of the LLNL organization on GitHub.

Note: RAJA core team members:

To facilitate testing contributions in PRs from forked repositories, we maintain a script to pull a PR branch from a forked repo into the RAJA repo. First, identify the number of the PR. Then, run the script from the top-level RAJA directory:

```
$ ./scripts/make_local_branch_from_fork_pr -b <PR #>
```

If successful, this will create a branch in your local copy of the RAJA repo labeled `pr-from-fork/<PR #>` and you will be on that local branch in your checkout space. To verify this, you can run the following command after you run the script:

```
$ git branch
```

You will see the new branch in the listing of branches and the branch you are on will be starred.

You can push the new branch to the RAJA repo on GitHub:

```
$ git push origin <branch-name>
```

and make a PR for the new branch. It is good practice to reference the original PR in the description of the new PR to track the original PR discussion and reviews.

All CI checks will be triggered to run on the new PR made in the RAJA repo. When everything passes and the PR is approved, it may be merged. When it is merged, the original PR from the forked repo will be closed and marked as merged unless it is referenced elsewhere, such as in a GitHub issue. If this is the case, then the original PR (from the forked repo) must be closed manually.

6.3.2 Continuous Integration (CI) Testing

The RAJA project employs multiple tools to run its tests for each GitHub *pull request*, all of which must pass before the pull request can be merged. These tools include:

- **Azure.** This runs builds for Linux, Windows, and MacOS environments using a variety of compilers. While we do GPU builds for CUDA, HIP, and SYCL on Azure, RAJA tests are run for each non-GPU build.
- **Appveyor.** This runs builds and tests for a Windows environment for two versions of the Visual Studio compiler.
- **Gitlab CI.** This runs builds and tests on platforms in the Livermore Computing *Collaboration Zone*. This is a recent addition for RAJA and is a work-in-progress to get full coverage of compilers and tests we need to exercise.

These tools integrate seamlessly with GitHub. They will automatically (re)run RAJA builds and tests as changes are pushed to each PR branch. Gitlab CI execution on Livermore Computing resources has some restrictions which are described below.

Gitlab CI support is still being developed to make it more easy to use with GitHub projects. The current state is described below.

Note: The status of checks (pass/fail, running status) for each of these tools can be viewed by clicking the appropriate link in the check section of a pull request.

Gitlab CI

If all memmbers of a GitHub project are members of the LLNL GitHub organization and have two-factor authentication enabled on their GitHub accounts, auto-mirroring on the Livermore Computing Collaboration Zone Gitlab server is enabled. Thus, Gitlab CI will run automatically for those projects on pull requests that are made by project members. Otherwise, due to Livermore Computing security policies, Gitlab CI must be launched manually by a *blessed* GitHub user satisfying the constraints described above. To manually initiate Gitlab CI on a pull request, add a comment with 'LGTM' in it.

It is important to note that RAJA shares its Gitlab CI workflow with other projects. See Shared Gitlab CI Workflow for more information.

Vetted Specs

The *vetted* compiler specs are those which we use during the RAJA Gitlab CI testing process. These can be viewed by looking at files in the RAJA `.gitlab` directory. For example,

```
$ ls -c1 .gitlab/*jobs.yml
.gitlab/lassen-jobs.yml
.gitlab/ruby-jobs.yml
```

lists the yaml files containing the Gitlab CI jobs for the lassen and ruby machines.

Then, executing a command such as:

```
$ git grep -h "SPEC" .gitlab/ruby-jobs.yml | grep "gcc"
    SPEC: "%gcc@4.9.3"
    SPEC: "%gcc@6.1.0"
    SPEC: "%gcc@7.3.0"
    SPEC: "%gcc@8.1.0"
```

will list the specs vetted on the ruby platform.

More details to come…

6.3.3 RAJA Build Configurations

RAJA must be built and tested with a wide range of compilers and with all of its supported back-ends. The project currently maintains two ways to build and test important configurations in a reproducible manner:

- **Build scripts.** The RAJA source repository contains a collection of simple build scripts that are used to generate build configurations for platforms in the Livermore Computing Center primarily.
- **Generated host-config files.** The RAJA repository includes a mechanism to generate *host-config* files (i.e., CMake cache files) using Spack.

Each of these specifies compiler versions and options, a build target (Release, Debug, etc.), RAJA features to enable (OpenMP, CUDA, etc.), and paths to required tool chains, such as CUDA, ROCm, etc. They are described briefly in the following sections.

RAJA Build Scripts

The build scripts in the RAJA `scripts` directory are used mostly by RAJA developers to quickly create a build environment to compile and run tests during code development.

Each script is executed from the top-level RAJA directory. THe scripts for CPU-only platforms require an argument that indicate the compiler version. For example,

```
$ ./scripts/lc-builds/toss3_clang.sh 10.0.1
```

Scripts for GPU-enabled platforms require three arguments: the device compiler version, followed by the compute architecture, followed by the host compiler version. For example,

```
$ ./scripts/lc-builds/blueos_nvcc_gcc.sh 10.2.89 sm_70 8.3.1
```

When a script is run, it creates a uniquely-named build directory in the top-level RAJA directory and runs CMake with arguments contained in the script to create a build environment in the new directory. One then goes into that directory and runs make to build RAJA, its tests, example codes, etc. For example,

```
$ ./scripts/lc-builds/blueos_nvcc_gcc.sh 10.2.89 sm_70 8.3.1
$ cd build_lc_blueos-nvcc10.2.89-sm_70-gcc8.3.1
$ make -j
$ make test
```

Eventually, these scripts may go away and be superceded by the Spack-based host-config file generation process when that achieves the level of compiler coverage that the scripts have.

Generated Host-Config Files

The RAJA repository contains two submodules uberenv and radiuss-spack-configs that work together to generate host-config files. These are projects in the GitHub LLNL organization and contain utilities shared by various projects. The main uberenv script can be used to drive Spack to generate a *host-config* file that contains all the information required to define a RAJA build environment. The host-config file can then be passed to CMake using the '-C' option to create a build configuration. *Spack specs* defining compiler configurations are maintained in files in the radiuss-spack-configs repository.

RAJA shares its uberenv workflow with other projects. The documentation for this is available in RADIUSS Uberenv Guide.

Generating a RAJA host-config file

This section describes the host-config file generation process for RAJA.

Machine specific configurations

Compiler configurations for Livermore computer platforms are contained in in sub-directories in the RAJA `scripts/uberenv/spack_configs` directory:

```
$ ls -c1 ./scripts/uberenv/spack_configs
blueos_3_ppc64le_ib
darwin
toss_3_x86_64_ib
blueos_3_ppc64le_ib_p9
config.yaml
```

To see currently supported configurations, please see the contents of the `compilers.yaml` file in each of these sub-directories.

Generating a host-config file

The main uberenv python script can be invoked from the top-level RAJA directory to generate a host-config file for a desired configuration. For example,

```
$ python ./scripts/uberenv/uberenv.py --spec="%gcc@8.1.0"
$ python ./scripts/uberenv/uberenv.py --spec="%gcc@8.1.0~shared+openmp tests=benchmarks"
```

Each command generates a corresponding host-config file in the top-level RAJA directory. The file name contains the platform and OS to which it applies, and the compiler and version. For example,

```
hc-quartz-toss_3_x86_64_ib-gcc@8.1.0-fjcjwd6ec3uen5rh6msdqujydsj74ubf.cmake
```

Specs that are exercised during the Gitlab CI process are found YAML files in the `RAJA/.gitlab` directory. See *Vetted Specs* for more information.

Building RAJA with a generated host-config file

To build RAJA with one of these host-config files, create a build directory and run CMake in it by passing the host-config file to CMake using the '-C' option. Then, run make and RAJA tests, if desired, to make sure the build was done properly:

```
$ mkdir <build dirname> && cd <build dirname>
$ cmake -C <path_to>/<host-config>.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

It is also possible to use the configuration with a RAJA CI script outside of the normal CI process:

```
$ HOST_CONFIG=<path_to>/<host-config>.cmake ./scripts/gitlab/build_and_test.sh
```

MacOS

In RAJA, the Spack configuration for MacOS contains the default compiler corresponding to the OS version (*compilers.yaml*), and a commented section to illustrate how to add *CMake* as an external package. You may install CMake with Homebrew, for example, and follow the process outlined above after it is installed.

Reproducing Docker Builds

RAJA uses docker container images that it shares with other LLNL GitHub projects for CI testing on GitHub. Currently, we use Azure for Linux, Windows, and MacOS builds and also have Appveyor builds for Windows.

You can reproduce these builds locally for testing with the following steps:

1. Run the command to build a local Docker image:

   ```
   $ DOCKER_BUILDKIT=1 docker build --target ${TARGET} --no-cache
   ```

   Here, ${TARGET} is replaced with one of the names following "AS" in the RAJA Dockerfile

2. To get dropped into a terminal in the Docker image, run the following:

   ```
   $ docker run -it axom/compilers:${COMPILER} /bin/bash
   ```

   Here, ${COMPILER} is replaced with the compiler you want (see the aforementioned Dockerfile).

Then, you can build, run tests, edit files, etc. in the Docker image. Note that the docker command has a '-v' argument that you can use to mount your local directory in the image; e.g., -v *pwd*:/opt/RAJA would mount the pwd as /opt/RAJA in the image.

6.3.4 RAJA Branch Development

Gitflow Branching Model

The RAJA project uses a simple branch development model based on Gitflow. The Gitflow model is centered around software releases. It is a simple workflow that makes clear which branches correspond to which phases of development. Those phases are represented explicitly in the branch names and structure of the repository. As in other branching models, developers develop code on a local branch and push their work to a central repository.

Persistent, Protected Branches

The **main** and **develop** branches are the two primary branches we use. They always exist and are protected in the RAJA GitHub project in that changes to them only occur as a result of approved pull requests. The distinction between the main and develop branches is an important part of Gitflow.

- The *main* branch records the release history of the project. Each time the main branch is changed, a new tag for a new code version is made. See *Semantic Versioning* for a description of the version numbering scheme we use.
- The *develop* branch is used to integrate and test new features and most bug fixes before they are merged into the main branch for a release.

Important: Development never occurs directly on the main branch or develop branch.

All other branches in the RAJA repo are temporary and are used to perform specific development tasks. When such a branch is no longer needed (e.g., after it is merged), the branch is deleted typically.

Feature Branches

Feature branches are created off of other branches (usually develop) and are used to develop new features, bug fixes, etc. before they are merged to develop and eventually main. *Feature branches are temporary*, living only as long as they are needed to complete development tasks they contain.

Each new feature, or other well-defined portion of work, is developed on its own feature branch. We typically include a label, such as “feature” or “bugfix”, in a feature branch name to make it clear what type of work is being done on the branch. For example, **feature/<name-of-feature>** for a new feature, **bugfix/<issue>** for a bugfix, etc.

Important: When doing development on a feature branch, it is good practice to regularly push your changes to the GitHub repository as a backup mechanism. Also, regularly merge the RAJA develop branch into your feature branch so that it does not diverge too much from other development on the project. This will help reduce merge conflicts that you must resolve when your work is ready to be merged into the RAJA develop branch.

When a portion of development is complete and ready to be merged into the develop branch, submit a *pull request* (PR) for review by other team members. When all issues and comments arising in PR review discussion have been addressed, the PR has been approved, and all continuous integration checks have passed, the pull request can be merged.

Important: Feature branches never interact directly with the main branch.

Other Important Branches

Release candidate and **hotfix** branches are two other important temporary branch types in Gitflow. They will be explained in the *RAJA Release Process* section.

Gitflow Illustrated

The figure below shows the basics of how branches interact in Gitflow.

6.3.5 RAJA Release Process

The RAJA release process typically involves the following sequence of steps:

1. Identify all work (features in development, outstanding PRs, etc.) to be to be included in the release.
2. Merge all PRs containing work to be included in the release into the develop branch.
3. Make a *Release Candidate Branch* from the develop branch. Finalize the release by completing remaining release tasks on that branch.
4. When the release candidate branch is ready, make a PR for it to be merged into the **main branch.** When it is approved and all CI checks pass, merge the release candidate branch into the RAJA main branch.

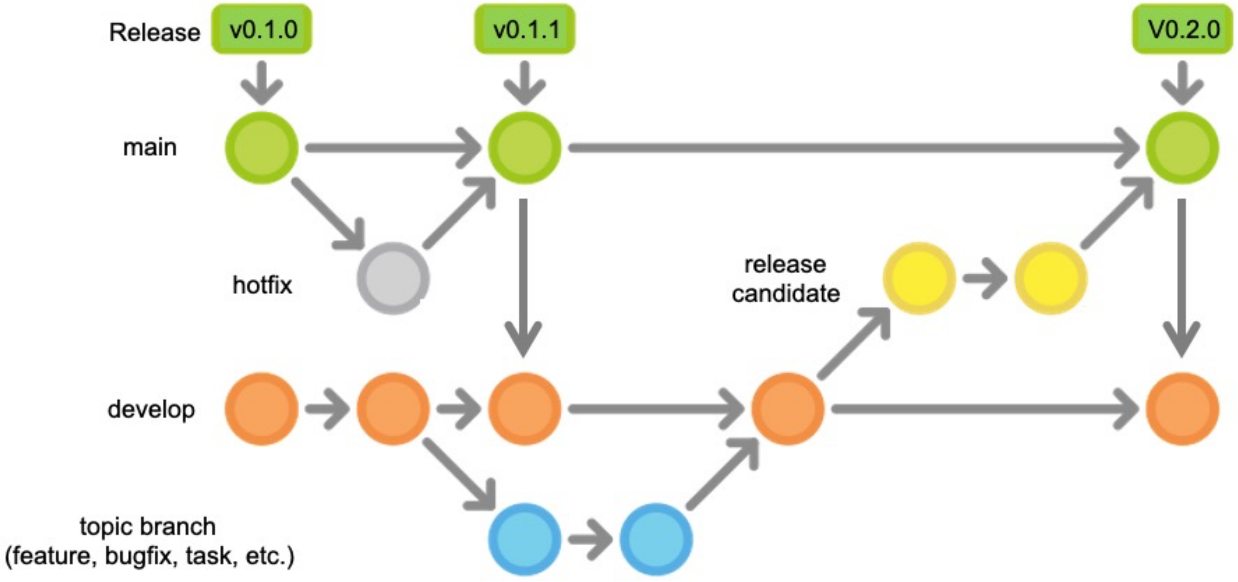

Fig. 7: This figure shows typical interactions between key branches in the Gitflow workflow. Here, development is shown following the v0.1.0 release. While development was ongoing, a bug was found and a fix was needed to be made available to users. A *hotfix* branch was made off of main and merged back to main after the issue was addressed. Release v0.1.1 was tagged and main was merged into develop so that it would not recur. Work started on a feature branch before the v0.1.1 release and was merged into develop after the v0.1.1 release. Then, a release candidate branch was made from develop. The release was finalized on that branch after which it was merged into main, and the v0.2.0 release was tagged. Finally, main was merged into develop.

5. On GitHub, make a new release with a tag for the release. Following our convention, the tag label should have the format `vMM.mm.pp`. See *Semantic Versioning* for a description of the version numbering scheme we use. In the GitHub release description, please note key features, bugfixes, etc. in the release. These should be a high-level summary of the contents of the `RELEASE_NOTES.md` file in the RAJA repo, which may contain more detailed information. Also, add a note to the release description to remind users to download the gzipped tarfile for the release instead of the assets GitHub creates for the release. The GitHub-created assets do not contain the RAJA submodules and will cause issues for users as a result.

   Important: For consistency, please follow a similar description pattern for all RAJA releases.

6. Check out the main branch locally and make sure it is up-to-date. Then, generate the release tarfile by running the script `./scripts/make_release_tarball.sh` from the top-level RAJA directory. If this is successful, a gzipped tarfile whose name includes the release tag **with no extraneous SHA-1 hash information** will be in the top-level RAJA directory.
7. Edit the release in GitHub and upload the tarfile to the release.
8. Make a PR to merge the main branch into the develop branch. After it passes all CI checks and is approved, merge the PR. This will ensure that all changes done to finalize the release will not be lost in future changes to the develop branch.

Release Candidate Branch

A *release candidate* branch is a temporary branch used to finalize a release. When the features, documentation, bug fixes, etc. to include in a release are complete and merged into the develop branch, a release candidate branch is made off of the develop branch. Typically, a release candidate branch is named **rc-<release #>.** Please see *RAJA Release Process* for a description of how a release candidate branch is used in the release process.

Important: Creating a release candidate branch starts the next release cycle whereby new work being performed on feature branches can be merged into the develop branch.

Finalizing a release on a release candidate branch involves the following steps:

1. If not already done, create a section for the release in the RAJA `RELEASE_NOTES.md` file. Describe all API changes, notable new features, bug fixes, improvements, build changes, etc. included in the release in appropriately labeled sections of the file. Please follow the pattern established in the file for previous releases. All changes that users should be aware of should be documented in the release notes. Hopefully, the release notes file has been updated along with the corresponding changes in PRs that are merged into the develop branch. At any rate, it is good practice to look over the commit history since the last release to ensure all important changes are captured in the release notes.
2. Update the version number entries for the new release in the `CMakeLists.txt` file in the top-level RAJA directory. These include entries for: `RAJA_VERSION_MAJOR`, `RAJA_VERSION_MINOR`, and `RAJA_VERSION_PATCHLEVEL`. These items are used to define corresponding macro values in the `include/RAJA/config.hpp` file when the code is built so that users can access and check the RAJA version in their code.
3. Update the `version` and `release` fields in the `docs/conf.py` file to the new release number. This information is used in the online RAJA documentation.

Important: No feature development is done on a release branch. Only bug fixes, release documentation, and other release-oriented changes are made on a release candidate branch.

Hotfix Branch

Hotfix branches are used in the (hopefully!) rare event that a bug is found shortly after a release and which has the potential to negatively impact RAJA users. A hotfix branch is used to address the issue and make a new release containing only the fixed code.

A hotfix branch is *made from main* with the name **hotfix/<issue>**. The issue is fixed (hopefully quickly!) and the release notes file is updated on the hotfix branch for the pending bugfix release. The branch is tested, against user code if necessary, to make sure the issue is resolved. Then, a PR is made to merge the hotfix branch into main. When it is approved and passes CI checks, it is merged into the main branch. Lastly, a new release is made in a fashion similar to the process described in *RAJA Release Process*. For completeness, the key steps for performing a hotfix release are:

1. Make a **hotfix** branch from main for a release (hotfix/<issue>), fix the issue on the branch and verify, testing against user code if necessary, and update the release notes file as needed.
2. When the hotfix branch is ready, make a PR for it to be merged into the **main branch.** When that is approved and all CI checks pass, merge it into the RAJA main branch.
3. On GitHub, make a new release with a tag for the release. Following our convention, the tag label should have the format `vMM.mm.ppp`. In the GitHub release description, note that the release is a bugfix release and describe the issue that is resolved. Also, add a note to the release description to download the gzipped tarfile for the release rather than one of the assets GitHub creates as part of the release.
4. Check out the main branch locally and make sure it is up-to-date. Then, generate the tarfile for the release by running the script `./scripts/make_release_tarball.sh` from the top-level RAJA directory. If this is successful, a gzipped tarfile whose name includes the release tag **with no extraneous SHA-1 hash information** will be in the top-level RAJA directory.

5. Make a PR to merge the main branch into the develop branch. After it passes all CI checks and is approved, merge the PR. This will ensure that changes for the bugfix will be included in future development.

6.3.6 Semantic Versioning

The RAJA project uses the *semantic* versioning scheme for assigning release numbers. Semantic versioning is a methodology for assigning a version number to a software release in a way that conveys specific meaning about code modifications from version to version. See Semantic Versioning for a more detailed description.

Version Numbers and Meaning

Semantic versioning is based on a three part version number *MM.mm.pp*:

- *MM* is the *major version number*. It is incremented when an incompatible API change is made. That is, the API changes in a way that may break code using an earlier release of the software with a smaller major version number.
- *mm* is the *minor version number*. It changes when functionality is added that is backward compatible, such as when the API grows to support new functionality yet the software will function the same as any earlier release of the software with a smaller minor version number when used through the intersection of two different APIs. The minor version number is always changed when the main branch changes, except possibly when the major version is changed; for example going from v1.0.0 to v2.0.0.
- *pp* is the *patch version number*. It changes when a bug fix is made that is backward compatible. That is, such a bug fix is an internal implementation change that fixes incorrect behavior. The patch version number is always changed when a hotfix branch is merged into main, or when changes are made to main that only contain bug fixes.

What Does a Change in Version Number Mean?

A key consideration in meaning for these three version numbers is that the software has a public API. Changes to the API or code functionality are communicated by the way the version number is incremented from release to release. Some important conventions followed when using semantic versioning are:

- Once a version of the software is released, the contents of the release **must not change**. If the software is modified, it **must** be released with a new version.
- A major version number of zero (i.e., *0.mm.pp*) is considered initial development where anything may change. The API is not considered stable.
- Version *1.0.0* defines the first stable public API. Version number increments beyond this point depend on how the public API changes.
- When the software is changed so that any API functionality becomes deprecated, the minor version number **must** be incremented, unless the major version number changes.
- A pre-release version may be indicated by appending a hyphen and a series of dot-separated identifiers after the patch version. For example, *1.0.1-alpha*, *1.0.1-alpha.1*, *1.0.2-0.2.5*.
- Versions are compared using precedence that is calculated by separating major, minor, patch, and pre-release identifiers in that order. Major, minor, and patch numbers are compared numerically from left to right. For example, 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1. When major, minor, and patch numbers are equal, a pre-release version number has lower precedence than none. For example, 1.0.0-alpha < 1.0.0.

By following these conventions, it is fairly easy to communicate intent of version changes to users and it should be straightforward for users to manage dependencies on RAJA.

6.4 RAJA Copyright and License Information

Copyright (c) 2016-22, Lawrence Livermore National Security, LLC.

Produced at the Lawrence Livermore National Laboratory.

All rights reserved. See additional details below.

Unlimited Open Source - BSD Distribution

LLNL-CODE-689114

OCEC-16-063

6.4.1 RAJA License

Copyright (c) 2016-2022, Lawrence Livermore National Security, LLC. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

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.