RAJA Documentation

Release 2022.10.5

LLNL

Mar 02, 2023

User Documentation

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 and Examples*. 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 should help get you up and running with RAJA quickly.

Requirements

The primary requirement for using RAJA is a C++14 standard compliant compiler. Certain features, such as various programming model back-ends like CUDA or HIP, msut be supported by the compiler you chose to use them. Available RAJA configuration options and how to enable or disable features are described in *Build Configuration Options*.

To build RAJA and use its most basic features, you will need:

- C++ compiler with C++14 support
- CMake version 3.23 or greater when building the HIP back-end, and version 3.20 or greater otherwise.

Get the Code

The RAJA project is hosted on GitHub: GitHub RAJA project. 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` option above is used to pull RAJA Git *submodules*, on which RAJA depends, into your local copy of the RAJA repository.

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

If you do not pass the `--recursive` argument to the `git clone` command, you can also type the following commands after cloning:

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

Either way, the end result is the same and you should be good to configure the code and build it.

Note:

- If you switch branches in a RAJA repo (e.g., you are on a branch, with everything up-to-date, and you run the command `git checkout <different branch name>`, you may need to run the command `git submodule update` to set the Git submodule versions to what is used by the new branch.
- If the set of submodules in a new branch is different than the previous branch you were on, you may need to run the command `git submodule update --init --recursive` to pull in the correct set of submodule and versions.

Dependencies

RAJA has several dependencies that are required based on how you want to build and use it. The RAJA Git repository has submodules that contain most of these dependencies.

RAJA includes other submodule dependencies, which are used to support our Gitlab CI testing. These are described in the RAJA Developer Guide.

Dependencies that are required to build the RAJA code are:

- A C++ 14 standard compliant compiler
- BLT build system
- CMake version 3.23 or greater when building the HIP back-end, and version 3.20 or greater otherwise.
- Camp compiler agnostic metaprogramming library

Other dependencies that users should be aware of that support certain features are:

- CUB CUDA utilities library, which is required for using the RAJA CUDA back-end.
- rocPRIM HIP parallel primitives library, which is required for using the RAJA HIP back-end.
- Desul, which is required if you want to use Desul atomics in RAJA instead of our current default atomics. Note that we plan to switch over to Desul atomics exclusively at some point.

Note: You may want or need to use external versions of camp, CUB, or rocPRIM instead of the RAJA submodules. This is usually the case when you are using RAJA along with some other library that also needs one of these. To do so, you need to use CMake variables to pass a path to a valid installation of each library. Specifically:

- External camp:

```
cmake \
... \
-Dcamp_DIR=path/to/camp/install \
...
```

- External CUB:

```
cmake \
... \
-DRAJA_ENABLE_EXTERNAL_CUB=On \
-DCUB_DIR=path/to/cub \
...
```

- External rocPRIM:

```
cmake \
... \
-DRAJA_ENABLE_EXTERNAL_ROCPRIM=On \
-DROCPRIM_DIR=path/to/rocPRIM \
...
```

More information about configuring GPU builds with CUDA or HIP is provided in *Additional RAJA Back-end Build Information*

Additional discussion of these dependencies, with respect to building RAJA, is provided in *Build and Install*. Other than that, you probably don't need to know much about them. If you are curious and want to know more, please click on the link to the library you want to know about in the above list.

Build and Install

The complexity of building and installing RAJA depends on which features you want to use and how easy it is to do this on your system.

Note: RAJA builds must be *out-of-source*. In particular, RAJA does not allow building in its source directory. You must create a build directory and run CMake in it.

RAJA uses CMake to configure a build. To create a "bare bones" configuration, build, and install it, you can do the following:

```
$ mkdir build-dir && cd build-dir
$ cmake -DCMAKE_INSTALL_PREFIX=/path/to/install ../
$ make  (or make -j <N> for a parallel build)
$ make install
```

Running `cmake` generates the RAJA build configuration. Running `make` compiles the code. Running `make install` copies RAJA header files to an `include` directory and installs the RAJA library in a `lib` directory, both in the directory location specified with the `-DCMAKE_INSTALL_PREFIX` CMake option.

Other build configurations are accomplished by passing other options to CMake. For example, if you want to use a C++ compiler other than the default on your system, you would pass a path to the compiler using the standard CMake option `-DCMAKE_CXX_COMPILER=path/to/compiler`. When you run CMake, it will generate output about the build configuration (compiler and version, options, etc.), which is helpful to make sure CMake is doing what you want. For a summary of RAJA configuration options, please see *Build Configuration Options*.

Note: RAJA is configured to build its tests, examples, and tutorial exercises by default. If you do not disable them with the appropriate CMake option (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 the corresponding test executables directly. They will be located in the `test` subdirectory in your build space. RAJA tests use the Google Test framework, so you can also run and filter tests via Google Test commands.

The source files for RAJA examples and exercises 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 your build space.

Additional RAJA Back-end Build Information

Configuring a RAJA build to support a GPU back-end, such as CUDA, HIP, or OpenMP target offload, typically requires additional CMake options, which we describe next.

CUDA

To run RAJA code on NVIDIA GPUs, one typically must have a CUDA compiler installed on the 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 as described earlier. The CUDA software stack and compiler are specified using the following CMake options:

- -DCUDA_TOOLKIT_ROOT_DIR=path/to/cuda/toolkit
- -DCMAKE_CUDA_COMPILER=path/to/nvcc

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

- CMAKE_CUDA_FLAGS_RELEASE
- CMAKE_CUDA_FLAGS_DEBUG
- CMAKE_CUDA_FLAGS_RELWITHDEBINFO

correspond to the standard CMake build types and are used to pass additional compiler options to nvcc.

Note: Often, nvcc must pass options to the host compiler, the arguments can be included using the `CMAKE_CUDA_FLAGS...` CMake variables listed above. Host compiler options must be prepended with the `-Xcompiler` directive to properly propagate.

To set the CUDA compute architecture, which should be chosen based on the NVIDIA GPU hardware you are using, you can use the `CMAKE_CUDA_ARCHITECTURES` CMake variable. For example, the CMake option `-DCMAKE_CUDA_ARCHITECTURES=70` will tell the compiler to use the *sm_70* SASS architecture in its second stage of compilation. The compiler 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 *Atomic Operations*.

- If you do not specify a value for `CMAKE_CUDA_ARCHITECTURES`, it will be set to *35* by default and CMake will emit a status message indicating this choice was made.
- If you give a `CMAKE_CUDA_ARCHITECTURES` value less than *35* (e.g., *30*), CMake will report this as an error and stop processing.

Also, RAJA relies on the CUB CUDA utilities library, mentioned earlier, for some CUDA back-end functionality. The CUB version included in the CUDA toolkit installation is used by default when available. This is the case for CUDA version 11 and later. RAJA includes a CUB submodule that is used by default with older versions of CUDA. To use an external CUB installation, provide the following options to CMake:

```
cmake \
... \
-DRAJA_ENABLE_EXTERNAL_CUB=On \
-DCUB_DIR=<path/to/cub> \
...
```

Note: The CUDA toolkit version of CUB is required for compatibility with the CUDA toolkit version of thrust starting with CUDA version 11.0.0. So, if you build RAJA with CUDA version 11 or higher, you should use the version of CUB contained in the CUDA toolkit version you are using to use Thrust and to be compatible with libraries that use Thrust.

Note: 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 a ROCm compiler and tool chain (which can also be used to compile code for NVIDIA GPUs, which is not covered in detail in RAJA user documentation).

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

Unlike CUDA, you do not specify a host compiler and a device compiler when using the AMD ROCm software stack. Typical CMake options to use when building with a ROCm stack are:

- -DROCM_ROOT_DIR=path/to/rocm

- -DHIP_ROOT_DIR=path/to/hip
- -DHIP_PATH=path/to/hip/binaries
- -DCMAKE_CXX_COMPILER=path/to/rocm/compiler

Additionally, you use the CMake variable `CMAKE_HIP_ARCHITECTURES` to set the target compute architecture. For example:

```
-DCMAKE_HIP_ARCHITECTURES=gfx908
```

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 options to CMake:

```
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 configure with an external CUB install using the CMake variables described in the CUDA section above.

OpenMP

To use OpenMP target offload GPU execution, additional options may need to be passed to the compiler. BLT variables are 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 \
  ... \
  -DBLT_OPENMP_COMPILE_FLAGS="-fopenmp;-fopenmp-targets=nvptx64-nvidia-cuda" \
  -DBLT_OPENMP_LINK_FLAGS="-fopenmp;-fopenmp-targets=nvptx64-nvidia-cuda" \
  ...
```

Compiler flags are passed to other compilers similarly, using flags specific to the compiler. Typically, the compile and link flags are the same as shown here.

RAJA Example Build Configuration Files

The RAJA repository has subdirectories `RAJA/scripts/*-builds` that contain 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.

Learning to Use RAJA

The RAJA repository contains a variety of example source codes that you are encouraged to view and run to learn about how to use RAJA:

- The `RAJA/examples` directory contains various examples that illustrate algorithm patterns.

- The `RAJA/exercises` directory contains exercises for users to work through along with complete solutions. These are described in detail in the *RAJA Tutorial and Examples* section.
- Other examples can also be found in the `RAJA/test` directories.

We mentioned earlier that RAJA examples, exercises, and tests are built by default when RAJA is compiled. So, unless you explicitly disable them when you run CMake to configure a RAJA build, you can run them after compiling RAJA. Executables for the examples and exercises will be located in the `<build-dir>/bin` directory in your build space. Test executables will be located in the `<build-dir>/test` directory.

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 and Examples*.

6.1.2 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)
```

6.1.3 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

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, when the CMake option `-DENABLE_TESTS=On` is used to enable tests in the build of an application that includes multiple CMake-based package builds, providing the CMake option `-DRAJA_ENABLE_TESTS=Off` will disable compilation of RAJA tests, while compiling them for other packages.
- 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 \
  -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 calling CMake. For example, to enable RAJA OpenMP functionality, pass the following argument to CMake:

```
cmake ... \
-DENABLE_OPENMP=On \
...
```

Alternatively, CMake options may be set in a CMake *cache file* and passing that file to CMake using the CMake `-C` option; for example:

```
cmake ... \
-C my_cache_file.cmake \
...
```

The directories `RAJA/scripts/*-builds` contain scripts that run CMake for various build configurations. These contain cmake invocations that use CMake cache files (we call them *host-config* files) and may be used as a guide for users trying to set their own options.

Next, we summarize RAJA CMake options and their defaults.

Available RAJA CMake 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_EXERCISES	On
(**RAJA_**)ENABLE_BENCHMARKS	Off
RAJA_ENABLE_REPRODUCERS	Off
(**RAJA_**)ENABLE_COVERAGE	Off (supported for GNU compilers only)

Other configuration options are available to specialize how RAJA is compiled:

Variable	Default
(**RAJA_**)ENABLE_WARNINGS_AS_ERRORS	Off
RAJA_ENABLE_FORCEINLINE_RECURSIVE	On (Intel compilers only)
RAJA_ALLOW_INCONSISTENT_OPTIONS	Off

RAJA Views/Layouts may be configured to check for out of bounds indexing at run time:

Variable	Default
RAJA_ENABLE_BOUNDS_CHECK	Off

Note: RAJA bounds checking is a run time check and will add considerable execution time overhead. Thus, this feature should only be used for debugging and correctness checking and should be disabled for production builds.

RAJA Features

Some RAJA features are enabled by RAJA-specific CMake variables.

Variable	Meaning
RAJA_ENABLE_RUNTIME_PLUGINS	Enable support for dynamically loaded RAJA plugins. Default is off.
RAJA_ENABLE_DESUL_ATOMICS	Replace RAJA atomic implementations with Desul variants at compile-time. Default is off.
RAJA_ENABLE_VECTORIZATION	Enable SIMD/SIMT intrinsics support. Default is on.

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	Off
(**RAJA**_)ENABLE_CUDA	Off
RAJA_ENABLE_CLANG_CUDA	Off
(**RAJA**_)ENABLE_HIP	Off
RAJA_ENABLE_TARGET_OPENMP	Off (when on, 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
RAJA_ENABLE_EXTERNAL_ROCPRIM	Off
RAJA_ENABLE_ROCTX	Off

Turning the `(RAJA_)ENABLE_CLANG_CUDA` variable on will build CUDA code with the native support in the Clang compiler.

Note: See *Getting Started With RAJA* for more information about using the `RAJA_ENABLE_EXTERNAL_CUB` and `RAJA_ENABLE_EXTERNAL_ROCPRIM` variables, as well other RAJA back-ends.

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

Data types, sizes, alignment, etc.

The options discussed in this section are typically not needed by users. They are provided for special cases when users want 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`.

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.)

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.

6.1.4 RAJA Features

The following sections describe the main RAJA features. They are intended to introduce users to the features and basic usage and also to provide a syntax reference guide. The sections contain links to RAJA tutorial materials that provide detailed examples of usage.

Elements of Loop Execution

The `RAJA::forall`, `RAJA::expt::dynamic_forall`, `RAJA::kernel`, and `RAJA::launch` template methods comprise the RAJA interface for kernel execution. `forall` methods execute simple, non-nested loops, `RAJA::kernel` methods support nested loops and other complex loop kernels and transformations, and

`RAJA::launch` creates an execution space in which kernels are written in terms of nested loops using the `RAJA::loop` method.

Note: The `forall` , `kernel`, and `launch` methods are in the `RAJA` namespace, while `dynamic_forall` is in the RAJA namespace for experimental features `RAJA::expt`. `RAJA::expt::dynamic_forall` will be moved to the `RAJA` namespace in a future RAJA release.

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

The following sections describe the basic aspects of these methods. Detailed examples showing how to use `RAJA::forall`, `RAJA::kernel`, `RAJA::launch` methods may be found in the *RAJA Tutorial and Examples*. Links to specific RAJA tutorial sections are provided in the sections below.

Simple Loops (RAJA::forall)

Consider a C-style loop that adds two vectors:

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

This may be written using `RAJA::forall` as:

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

A `RAJA::forall` loop execution method is a template that takes an *execution policy* type template parameter. A `RAJA::forall` method takes two arguments: an iteration space object, such as a contiguous range of loop indices as shown here, and a single lambda expression representing the loop kernel 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. Please see *Indices, Segments, and IndexSets* for details about RAJA iteration spaces.

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.

As an extension of `RAJA::forall`, the `RAJA::expt::dynamic_forall` method enables users to compile using a list of execution policies and choose the execution policy at run-time. For example, a user may want to have N policies available and at run-time choose which policy to use:

```
using exec_pol_list = camp::list<pol_0,
                                  ...
                                 pol_N>;
int pol = i; //run-time value

RAJA::expt::dynamic_forall<exec_pol_list>(pol, RAJA::TypedRangeSegment<int>(0, N),␣
↪[=] (int i) {
   c[i]  = a[i] + b[i];
});
```

While static loop execution using `forall` methods is a subset of `RAJA::kernel` functionality, described next, we maintain the `forall` interfaces 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.

Please see the following tutorial sections for detailed examples that use `RAJA::forall`:

- *Basic Loop Execution: Vector Addition*
- *Sum Reduction: Vector Dot Product*
- *Reduction Types and Kernels with Multiple Reductions*
- *Atomic Operations: Computing a Histogram*
- *Iteration Spaces: Segments and IndexSets*
- *Iteration Space Coloring: Mesh Vertex Sum*
- *Permuted Layout: Batched Matrix-Multiplication*

Complex Loops (RAJA::kernel)

A `RAJA::kernel` template provides ways to compose and execute arbitrary loop nests and other complex kernels. The `RAJA::kernel` interface employs similar concepts to `RAJA::forall` but extends it to support much more complex kernel structures. Each `RAJA::kernel` method is a template that takes an *execution policy* type template parameter. The execution policy can be an arbitrarily complex sequence of nested templates that define a kernel execution pattern. In its simplest form, `RAJA::kernel` takes two arguments: a *tuple* of iteration space objects, and a lambda expression representing the kernel inner loop body. In more complex usage, `RAJA::kernel` can take multiple lambda expressions representing different portions of the loop kernel body.

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
     }
}
```

It is important to note that we do not recommend writing a RAJA version of this by nesting `RAJA::forall` statements. For example:

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

This would work for some execution policy choices, but not in general. Also, this approach treats each loop level as an independent entity, which 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 using nested ‘‘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 enables the ability to transform and apply different parallel execution patterns by changing the execution policy type and **not the kernel code**, in many cases.

The C-style loop above nest may be written using `RAJA::kernel` 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::TypedRangeSegment<int>(0, NN),
                   ...,
                   RAJA::TypedRangeSegment<int>(0, N0),

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

);
```

In the case we discuss here, the execution policy contains a nested sequence of `RAJA::statement::For` types, indicating an iteration over each level in the loop nest. Each of these statement types takes three template parameters:

- an integral index parameter that binds the statement to the item in the iteration space tuple corresponding to that index
- an execution policy type for the associated loop nest level
- 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 a 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 `RAJA::statement::Lambda` statements. The simplest form, shown above, requires that each lambda expression passed to a `RAJA::kernel` method **must take an index argument for each iteration space.** With this type of lambda statement, the entire iteration space must be active in a surrounding `For` construct. A compile time `static_assert` will be triggered if any of the arguments are undefined, indicating that something is not correct.

A second `RAJA::statement::Lambda` type, which is an extension of the first, takes additional template parameters which specify which iteration spaces 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 indices from which elements in the segment tuple are passed as arguments to the lambda, and in which argument order. Here, we pass all segment indices so the lambda kernel body definition could be identical to on passed to the previous RAJA version. RAJA offers other types such as `RAJA::Offsets`, and `RAJA::Params` to identify offsets and parameters in segments and parameter tuples that could be passed to `RAJA::kernel` methods. See *Matrix Multiplication: RAJA::kernel* for an example.

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.

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*.

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

Please see the following tutorial sections for detailed examples that use `RAJA::kernel`:

- *Basic RAJA::kernel Mechanics and Nested Loop Ordering*
- *RAJA::kernel Execution Policies*

- *Matrix Transpose*
- *OffsetLayout: Five-point Stencil*
- *Matrix Multiplication: RAJA::kernel*

Hierarchical loops (RAJA::launch)

The `RAJA::launch` template is an alternative interface to `RAJA::kernel` that may be preferred for certain types of complex kernels or based on coding style preferences.

`RAJA::launch` optionally allows either host or device execution to be chosen at run time. The method takes an execution policy type that will define the execution environment inside a lambda expression for a kernel to be run on a host, device, or either. Kernel algorithms are written inside main lambda expression using `RAJA::loop` methods.

The `RAJA::launch` framework aims to unify thread/block based programming models such as CUDA/HIP/SYCL while maintaining portability on host back-ends (OpenMP, sequential). As we showed earlier, when using the `RAJA::kernel` interface, developers express all aspects of nested loop execution in an 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. For example, here is an example of a `RAJA::launch` kernel that copies values from an array in into a *shared memory* array:

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

  RAJA::loop<team_x> (ctx, RAJA::RAJA::TypedRangeSegment<int>(0, teamRange), [&] (int␣
↪bx) {

    RAJA_TEAM_SHARED double s_A[SHARE_MEM_SIZE];

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

      ctx.teamSync();

 )};

});
```

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

- `RAJA::launch` template. This creates an execution environment in which a kernel implementation is written using nested `RAJA::loop` statements. The launch policy template parameter used with the `RAJA::launch` method enables specification of both a host and device execution environment, which enables run time selection of kernel execution.
- `RAJA::LaunchParams` type. This type takes a number of teams and and a number of threads as arguments.
- `RAJA::loop` template. These are used to define hierarchical parallel execution of a kernel. Operations within a loop are mapped to either teams or threads based on the execution policy template parameter provided.

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, while thread policies are aliases for RAJA GPU thread policies in the x,y,z dimensions. In a host execution environment, teams and threads may be mapped to sequential loop execution or OpenMP threaded regions. Often, the RAJA::LaunchParams method can take an empty argument list for host execution.

Please see the following tutorial sections for detailed examples that use RAJA::launch:

- *RAJA::Launch Basics*
- *RAJA::Launch Execution Policies*
- *Matrix Transpose*

Multi-dimensional 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 those described in *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) {
       \\ 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 RAJA::make_CombingingAdapter template method takes a lambda expression for the loop body and an arbitrary number of index arguments. It provides a *flattened* iteration space via the getRange method that can be passed as the iteration space to the RAJA::forall method, for example. 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::TypedRangeSegment segments.

Policies

RAJA kernel execution methods take an execution policy type template parameter to specialize execution behavior. Typically, the policy indicates which programming model back-end to use and other information about the execution pattern, such as number of CUDA threads per thread block, whether execution is synchronous or asynchronous, etc. This section describes RAJA policies for loop kernel execution, scans, sorts, reductions, atomics, etc. Please detailed examples in *RAJA Tutorial and Examples* for a variety of use cases.

As RAJA functionality evolves, new policies are 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 run time.

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)'
omp_parallel_collapse_exec	kernel (Collapse + ArgList)	Use in Collapse statement to parallelize multiple loop levels in loop nest indicated using ArgList

Important: **RAJA only provides a nowait policy option for static scheduling** since that is the only schedule case that can be used with nowait and be correct in general when executing 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.

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 use with the RAJA TBB back-end, which supports a subset of RAJA features.

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 run time.

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	Local_Array (Loop)	Reshape 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	Local_Array (Loop)	Creates a unique thread id for each thread on x-dimension of the grid (expt namespace)
cuda/hip_global_thread_y	Local_Array (Loop)	Same as above, but uses threads in y-dimension (expt namespace)
cuda/hip_global_thread_z	Local_Array (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<MaskType>	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 1024 threads per block. Attempting to execute a kernel with more than the maximum allowed 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>	kernel (For)	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 by passing an IndexSet to a `RAJA::forall` method, for example, 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 >
```

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 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>([=]() {
```

(continues on next page)

(continued from previous page)

```
  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. So they should not be used for kernels containing reductions.

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/sycl_atomic	any CUDA/HIP/SYCL policy	Atomic operation performed in a CUDA/HIP/SYCL 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/SYCL 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::TypedRangeSegment<int> seg(0, N),␣
↪kernel);

RAJA::forall< RAJA::omp_parallel_for_exec >(RAJA::TypedRangeSegment<int> 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::TypedRangeSegment<int> 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 since reductions are not supported for the RAJA TBB back-end.
- 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 discussed in *Complex Loops (RAJA::kernel)*. A `RAJA::statement::For<ArgID, ExecPolicy, Enclosed Satements>` type indicates a for-loop structure. The `ArgID` parameter is an integral constant that identifies the position of the iteration space in the iteration space tuple passed to the `RAJA::kernel` method to be used for the loop. The `ExecPolicy` is the RAJA execution policy to use on the loop, which is similar to `RAJA::forall` usage. The `EnclosedStatements` type is a nested template parameter that contains whatever is needed to execute the kernel and which forms a valid StatementList. The `RAJA::statement::Lambda<LambdaID>` type invokes the lambda expression corresponding to its position 'LambdaID' in the sequence of lambda expressions in the `RAJA::kernel` argument list. For example, a simple sequential for-loop:

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

can be represented using the RAJA kernel interface as:

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

RAJA::kernel<KERNEL_POLICY>(
  RAJA::make_tuple(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 the `RAJA::kernel` examples in *RAJA Tutorial and Examples*.

Note: All of the statement types described below are in the namespace `RAJA::statement`. For brevity, we omit the namespaces in the discussion in this section.

Note: `RAJA::kernel_param` functions similarly 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.

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 launch 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. Warp sync is not supported in HIP, so the HIP variant is a no-op.

Statement types that launch 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 multithreaded 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 auxilliary types used in the above statements. 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 parameter 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 (RAJA::kernel)*.

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.

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

Please see the following tutorial sections for detailed examples that use RAJA iteration space concepts:

- *Iteration Spaces: Segments and IndexSets*
- *Iteration Space Coloring: Mesh Vertex Sum*

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.

Segments and IndexSets

A RAJA **Segment** represents a set of indices that one wants to execute as a unit for a kernel. RAJA provides the following Segment types:

- `RAJA::TypedRangeSegment` represents a stride-1 range
- `RAJA::TypedRangeStrideSegment` represents a (non-unit) stride range
- `RAJA::TypedListSegment` represents an arbitrary set of indices

A `RAJA::TypedIndexSet` is a container that can hold an arbitrary collection of segments to compose iteration patterns in a single kernel invocation.

Segment and IndexSet types are used in `RAJA::forall` and other RAJA kernel execution mechanisms to define the iteration space for a kernel.

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 segments. 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.

Please see *Iteration Spaces: Segments and IndexSets* for a detailed discussion of how to create and use these segment types.

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 kernel execution templates.

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`, `RAJA::Layout`, and `RAJA::OffsetLayout` classes.

Please see the following tutorial sections for detailed examples that use RAJA Views and Layouts:

- *Data Views and Layouts*
- *OffsetLayout: Five-point Stencil*
- *Permuted Layout: Batched Matrix-Multiplication*
- *RAJA::kernel Execution Policies*
- *RAJA::Launch Execution Policies*

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);

  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[10];
```

(continues on next page)

(continued from previous page)

```
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 < 5; ++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 ... 4
```

will index into the data array as:

```
0 1 2 ... 9
```

The arguments to the `RAJA::make_offset_layout` method are C++ arrays that hold the begin-end values of indices in the half-open interval :math:[begin, end)'. 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 3, 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 *OffsetLayout: Five-point Stencil* and *Permuted Layout: Batched Matrix-Multiplication* 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 accessing 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)
```

(continues on next page)

(continued from previous page)

```
// 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 checking 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 kernels launched using `RAJA::forall`, `RAJA::kernel`, and `RAJA::launch` methods in a portable, thread-safe manner. Users may use as many reduction objects in a loop kernel as they need. Available RAJA reduction types are described in this section.

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 compatible with the execution policy used by the kernel in which it is used.** 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.

Please see the following tutorial sections for detailed examples that use RAJA reductions:

- *Reduction Types and Kernels with Multiple Reductions*.

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...
//
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];

});

// 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*.

Experimental Reduction Interface

An experimental reduction interface is now available that offers several usability and performance advantages over the current reduction model in RAJA. The new interface allows `RAJA::forall` to take optional "plugin-like" objects to extend the execution behavior of a `RAJA::forall` execution context.

The new interface passes `RAJA::expt::Reduce<OP_TYPE>` objects as function arguments to `RAJA::forall` and provides users with thread-local variables of the reduction data type to be updated inside the lambda. This differs from the current reduction model in which `RAJA::ReduceOP<REDUCE_POL, T>` objects are captured by the user-supplied kernel body lambda expression.

RAJA::expt::Reduce

```
double* a = ...;
```

(continues on next page)

(continued from previous page)

```
double rs = 0.0;
double rm = 1e100;

RAJA::forall<EXEC_POL> ( Res, Seg,
RAJA::expt::Reduce<RAJA::operators::plus>(&rs),
RAJA::expt::Reduce<RAJA::operators::minimum>(&rm),
[=] (int i, double& _rs, double& _rm) {
  _rs += a[i];
  _rm = RAJA_MIN(a[i], _rm);
}
);

std::cout << rs ...
std::cout << rm ...
```

- Each `RAJA::expt::Reduce` argument to `RAJA::forall` is templated on a reduction operator, and takes a pointer to a target variable to write the final reduction result to, `&rs` and `&rm` in the example code above. The reduction operation will include the existing value of the given target variable.
- The kernel body lambda expression passed to `RAJA::forall` must have a parameter corresponding to each `RAJA::expt::Reduce` argument, `_rs` and `_rm` in the example code. These parameters refer to a local target for each reduction operation. It is important to note that the parameters follow the kernel iteration variable, `i` in this case, and appear in the same order as the corresponding `RAJA::expt::Reduce` arguments to `RAJA::forall`. The parameters' types must be references to the types used in the `RAJA::expt::Reduce` arguments.
- The local variables referred to by `_rs` and `_rm` are initialized with the *identity* of the reduction operation to be performed.
- The local variables are updated in the user supplied lambda.
- The local variables are reduced to a single value, combining their values across all threads participating in the `RAJA::forall` execution.
- Finally, the target variable is updated with the result of the `RAJA::forall` reduction by performing the reduction operation to combine the existing value of the target variable and the result of the `RAJA::forall` reduction.
- The final reduction value is accessed by referencing the target variable passed to `RAJA::expt::Reduce` in the `RAJA::forall` method.

Note: In the above example `Res` is a resource object that must be compatible with the `EXEC_POL`. `Seg` is the iteration space object for `RAJA::forall`.

Important: The order and types of the local reduction variables in the kernel body lambda expression must match exactly with the corresponding `RAJA::expt::Reduce` arguments to the `RAJA::forall` to ensure that the correct result is obtained.

RAJA::expt::ValLoc

As with the current RAJA reduction interface, the new interface supports *loc* reductions, which provide the ability to get a kernel/loop index at which the final reduction value was found. With this new interface, *loc* reductions are performed using `ValLoc<T>` types. Since they are strongly typed, they provide `min()` and `max()` operations that

are equivalent to using `RAJA_MIN()` or `RAJA_MAX` macros as demonstrated in the code example below. Users must use the `getVal()` and `getLoc()` methods to access the reduction results:

```
double* a = ...;

using VL_DOUBLE = RAJA::expt::ValLoc<double>;
VL_DOUBLE rm_loc;

RAJA::forall<EXEC_POL> ( Res, Seg,
RAJA::expt::Reduce<RAJA::operators::minimum>(&rm_loc),
[=] (int i, VL_DOUBLE& _rm_loc) {
  _rm_loc = RAJA_MIN(VL_DOUBLE(a[i], i), _rm_loc);
  //_rm_loc.min(VL_DOUBLE(a[i], i)); // Alternative to RAJA_MIN
}
);

std::cout << rm_loc.getVal() ...
std::cout << rm_loc.getLoc() ...
```

Lambda Arguments

This interface takes advantage of C++ parameter packs to allow users to pass any number of `RAJA::expt::Reduce` objects to the `RAJA::forall` method:

```
double* a = ...;

using VL_DOUBLE = RAJA::expt::ValLoc<double>;
VL_DOUBLE rm_loc;
double rs;
double rm;

RAJA::forall<EXEC_POL> ( Res, Seg,
  RAJA::expt::Reduce<RAJA::operators::plus>(&rs),         // --> 1 double added
  RAJA::expt::Reduce<RAJA::operators::minimum>(&rm),      // --> 1 double added
  RAJA::expt::Reduce<RAJA::operators::minimum>(&rm_loc), // --> 1 VL_DOUBLE added
  RAJA::expt::KernelName("MyFirstRAJAKernel"),            // --> NO args added
  [=] (int i, double& _rs, double& _rm, VL_DOUBLE& _rm_loc) {
    _rs += a[i];
    _rm = RAJA_MIN(a[i], _rm);
    _rm_loc.min(VL_DOUBLE(a[i], i));
  }
);

std::cout << rs ...
std::cout << rm ...
std::cout << rm_loc.getVal() ...
std::cout << rm_loc.getLoc() ...
```

Again, the lambda expression parameters are in the same order as the `RAJA::expt::Reduce` arguments to `RAJA::forall`. Both the types and order of the parameters must match to get correct results and to compile successfully. Otherwise, a static assertion will be triggered:

```
LAMBDA Not invocable w/ EXPECTED_ARGS.
```

Note: This static assert is only enabled when passing an undecorated C++ lambda. Meaning, this check will not

happen when passing extended-lambdas (i.e. DEVICE tagged lambdas) or other functor like objects.

Note: The experimental `RAJA::forall` interface is more flexible than the current implementation, other optional arguments besides `RAJA::expt::Reduce` can be passed to a `RAJA::forall` to extend its behavior. In the above example we demonstrate using `RAJA::expt::KernelName`, which wraps a `RAJA::forall` executing under a `HIP` or `CUDA` policy in a named region. Use of `RAJA::expt::KernelName` does not require an additional parameter in the lambda expression.

Atomic Operations

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.

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

Note: Each RAJA atomic operation is templated on an *atomic policy* type, which **must be compatible with the execution policy used by the kernel in which it is used.** For example, in a CUDA kernel, a CUDA atomic policy type must be used.

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

Note: RAJA support for CUDA atomic operations may be specific to the compute architecture for which the code is compiled. Please see *CUDA Atomics Architecture Dependencies* for more information.

RAJA currently supports two different implementations of atomic operations via the same basic interface. The default implementation is the original one developed in RAJA and which has been available for several years. Alternatively, one can choose an implementation based on DESUL at compile time. Please see *DESUL Atomics Support* for more information. Eventually, we plan to deprecate the original RAJA implementation and provide only the DESUL implementation. The RAJA atomic interface is expected to change when we switch over to DESUL atomic support. Specifically, the atomic policy noted above will no longer be used.

Please see the following tutorial sections for detailed examples that use RAJA atomic operations:

- *Atomic Operations: Computing a Histogram*.

Atomic Operations

RAJA atomic support the most common atomic operations.

Note: Each atomic method described below returns the value of the potentially modified argument (i.e., *acc) immediately before the atomic operation is applied, in case a user requires it.

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::TypedRangeSegment<int>(0, N),
  [=] RAJA_DEVICE (int i) {
```

(continues on next page)

(continued from previous page)

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

});
```

After this kernel executes, the value reference by 'sum' will be 'N'.

AtomicRef

RAJA also provides an 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.

CUDA Atomics Architecture Dependencies

The implementations for RAJA atomic operations may vary depending on which CUDA architecture is available and/or specified when 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 the ability to use DESUL Atomics as an alternative to the default implementation of RAJA atomics. DESUL atomics are considered an **experimental** feature in RAJA at this point and may impact the performance of some atomic functions. While DESUL atomics typically yields better or similar performance to RAJA default atomics, some atomic operations may perform worse when using DESUL.

To enable DESUL atomics, pass the option to CMake when configuring a RAJA build: `-DRAJA_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 for the most part. The exception to this is when RAJA atomic helper functions are used instead of the backward-compatible API functions specified by *Atomic Operations*. By *helper functions*, we mean the RAJA atomic methods which take an atomic policy object as the first argument, instead of specifying the atomic 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 `RAJA::cuda_atomic` or `RAJA::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 when we switch to use DESUL atomics exclusively and remove the default RAJA atomic operations.

Scan Operations

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. Please see *Policies* for more information.
- 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 or HIP back-end, RAJA implementation uses the NVIDIA CUB library or AMD rocPRIM library, respectively. Typically, the CMake variable `CUB_DIR` or `ROCPRIM_DIR` will be automatically set to the location of the CUB or rocPRIM library for the CUDA or rocPRIM installation specified when either back-end is enabled. More details for configuring the CUB or rocPRIM library for a RAJA build can be found in *Dependencies*.

Please see the following tutorial sections for detailed examples that use RAJA scan operations:

- *Parallel Scan Operations*.

Scan Operations

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

An **inclusive scan** takes the input sequence

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

and calculates the output sequence:

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

using the recursive definition

$$y_0 = x_0$$

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

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.,

$y_0 = op_{identity}$

$y_i = y_{i-1}$ 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)`. This is shown in the examples in *Parallel Scan Operations*.

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`.

Sort Operations

RAJA provides portable parallel sort operations, which 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. Please see *Policies* for more information.
- 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 a sequence sorted in **non-decreasing** order.

Also:

Note: For sorts using the CUDA or HIP back-end, RAJA implementation uses the NVIDIA CUB library or AMD rocPRIM library, respectively. Typically, the CMake variable `CUB_DIR` or `ROCPRIM_DIR` will be automatically set to the location of the CUB or rocPRIM library for the CUDA or rocPRIM installation specified when either back-end is enabled. More details for configuring the CUB or rocPRIM library for a RAJA build can be found *Dependencies*.

Please see the following tutorial sections for detailed examples that use RAJA scan operations:

- *Parallel Sort Operations*

Sort Operations

In general, a sort operation takes a sequence of numbers 'x' and a binary comparison operator 'op' to form a strict weak ordering of elements in input sequence 'x' and produce 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 means '!(a op b) && !(b op a)' is true.

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.

$x = \{ a_0, b_0, a_1, \dots \}$

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:

$y = \{ a_0, a_1, b_0, \dots \}$

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

$y = \{ a_0, a_1, b_0, \dots \}$

or

$y = \{ a_1, a_0, b_0, \ldots \}$

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 integer 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:

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

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 *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 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 the default RAJA less-than comparator.

Equivalently, the `RAJA::operators::less` comparator operator could be passed as the second argument to the sort routine to produce the same result:

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

Note that container arguments can be generated from iterators using `RAJA::make_span(out, N)`, where we pass the base pointer for the array and its length.

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. Detailed examples are provided in *Parallel Sort Operations*.

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

RAJA Stable Sorts

RAJA stable sort operations are used 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`.

Resources

This section describes the basic concepts of resource types and how to use them with RAJA-based kernels using `RAJA::forall`, `RAJA::kernel`, ‘‘ RAJA::launch‘‘, etc. Resources are used as an interface to various RAJA back-end constructs and their respective hardware. Currently there exist resource types for `Cuda`, `Hip`, `Omp` (target) and `Host`. Resource objects allow one to allocate and deallocate storage in memory spaces associated with RAJA back-ends and copy data between memory spaces. They also allow one to execute RAJA kernels asynchronously on a respective thread/stream. Resource support in RAJA is rudimentary at this point and its functionality / behavior may change as it is developed.

Note:

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

Each resource has a set of underlying functionality that is synonymous across each resource type.

Methods	Brief description
get_platform	Returns the underlying camp platform associated with the resource.
get_event	Return an event object for the resource from the last resource call.
allocate	Allocate data on a resource back-end.
deallocate	Deallocate data on a resource back-end.
memcpy	Perform a memory copy from a source location to a destination location on a resource back-end.
memset	Set memory value in an allocation on a resource back-end.
wait	Wait for all operations enqueued on a resource to complete before proceeding.
wait_for	Enqueue a wait on a resource stream/thread for a user passed event to complete.

Note: `deallocate`, `memcpy` and `memset` operations 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 back-end information/functionality. For example, each CUDA resource contains a `cudaStream_t` value with an associated get method. The basic interface for each resource type is summarized in Camp resource.

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

Type-Erasure

Resources can be declared in two ways, as a type-erased resource or as a concrete resource. The underlying run time functionality is the same for both.

Here is one way to construct a concrete CUDA resource type:

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

A type-erased resource allows a user the ability to change the resource back-end at run time. For example, to choose a CUDA GPU device resource or host resource at run time, one could do the following:

```
RAJA::resources::Resource* my_res = nullptr;

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

When `use_gpu` is true, `my_res` will be a CUDA GPU device resource. Otherwise, it will be a host CPU resource.

Memory Operations

The example discussed in this section illustrates most of the memory operations that can be performed with A common use case for a resource is to manage arrays in the appropriate memory space to use in a kernel. Consider the following code example:

```
// create a resource for a host CPU and a CUDA GPU device
RAJA::resources::Resource host_res{RAJA::resources::Host()};
RAJA::resources::Resource cuda_res{RAJA::resources::Cuda()};

// allocate arrays in host memory and device memory
int N = 100;

int* host_array = host_res.allocate<int>(N);
int* gpu_array = cuda_res.allocate<int>(N);

// initialize values in host_array....

// initialize gpu_array values to zero
cuda_res.memset(gpu_array, 0, sizeof(int) * N);
```

(continues on next page)

(continued from previous page)

```
// copy host_array values to gpu_array
cuda_res.memcpy(gpu_array, host_array, sizeof(int) * N);

// execute a CUDA kernel that uses gpu_array data
RAJA::forall<RAJA::cuda_exec<128>>(RAJA::TypedRangeSegment<int>(0, N),
  [=] RAJA_DEVICE(int i) {
    // modify values of gpu_array...
  }
);

// copy gpu_array values to host_array
cuda_res.memcpy(host_array, gpu_array, sizeof(int) * N);

// do something with host_array on CPU...

// de-allocate array storage
host_res.deallocate(host_array);
cuda_res.deallocate(gpu_array);
```

Here, we create a CUDA GPU device resource and a host CPU resource and use them to allocate an array in GPU memory and one in host memory, respectively. Then, after initializing the host array, we use the CUDA resource to copy the host array to the GPU array storage. Next, we run a CUDA device kernel which modifies the GPU array. After using the CUDA resource to copy the GPU array values into the host array, we can do something with the values generated in the GPU kernel on the CPU host. Lastly, we de-allocate the arrays.

Kernel Execution and Resources

Resources can be used with the following RAJA kernel execution interfaces:

- `RAJA::forall`
- `RAJA::kernel`
- `RAJA::launch`
- `RAJA::sort`
- `RAJA::scan`

Although we show examples using mainly `RAJA::forall` in the following discussion, resource usage with the other methods listed is similar and provides similar behavior.

Usage

Specifically, a resource can be passed optionally as the first argument in a call to one of these methods. For example:

```
RAJA::forall<ExecPol>(my_res, .... );
```

Note: When a resource is not passed when calling one of the methods listed above, the *default* resource type associated with the execution policy is used in the internal implementation.

When passing a CUDA or HIP resource, the method will execute asynchronously on a GPU stream. Currently, CUDA and HIP are the only resource types that enable asynchronous threading.

Note: Support for OpenMP CPU multithreading, which would use the `RAJA::resources::Host` resource type, and OpenMP target offload which would use the `RAJA::resources::Omp` resource type, is incomplete and under development.

The resource type passed to one of the methods listed above must be a concrete type; i.e., not type erased. The reason is that this allows consistency checking via a compile-time assertion to ensure that the passed resource is compatible with the given execution policy. For example:

```
using ExecPol = RAJA::cuda_exec_async<BLOCK_SIZE>;

RAJA::resources::Cuda my_cuda_res;
RAJA::forall<ExecPol>(my_cuda_res, .... ); // Successfully compiles

RAJA::resources::Resource my_res{RAJA::resources::Cuda()};
RAJA::forall<ExecPol>(my_res, .... )       // Compilation error since resource type␣
↪is not concrete

RAJA::resources::Host my_host_res;
RAJA::forall<ExecPol>(my_host_res, .... )  // Compilation error since resource type␣
↪is incompatible with the execution policy
```

IndexSet Usage

Recall that a kernel that uses a RAJA IndexSet to describe the kernel iteration space, require a two execution policies (see *RAJA IndexSet Execution Policies*). Currently, a user may only pass a single resource to a method taking an IndexSet argument. The resource is used for the *inner* execution over each segment in the IndexSet, not for the *outer* iteration over segments. For example:

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

Default Resources

When a resource is not provided by the user, a *default* resource that corresponds to the execution policy is used. The default resource can be accessed in multiple 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 in two different ways from an execution policy:

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

Or:

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

Note: For CUDA and HIP, the default resource is *NOT* associated with the default CUDA or HIP stream. It is its own stream defined by the underlying camp resource. This is intentional to break away from some issues that arise from the synchronization behavior 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();
```

This wait call will block all execution until all operations enqueued on a resource complete.

Alternatively, a user can enqueue the event on a specific resource, forcing only the resource to wait for the operation associated with the event to complete:

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

All methods listed above near the beginning of the RAJA resource discussion return an event object so users can access the event associated with the method call. This allows one to set up dependencies between resource objects and operations, as well as define and control asynchronous execution patterns.

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:

```
RAJA::forall<cuda_exec_async<BLOCK_SIZE>>(my_cuda_res, ...);
```

will *not* generate a cudaStreamEvent, whereas:

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

will generate a cudaStreamEvent.

Example

The example presented here executes three kernels across two CUDA streams on a GPU with a requirement that the first and second kernel finish execution before the third is launched. It also shows copying memory from the device to host on a resource that we described earlier.

First, we define two concrete CUDA resources and one concrete host resource, and define an asynchronous CUDA execution policy type:

```
  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, we allocate data for two GPU arrays and one host array, all of length 'N':

```
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, we launch a GPU kernel on the CUDA stream associated with the resource `res_gpu1`, without keeping a handle to the associated event:

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

Next, we execute another GPU kernel on the CUDA stream associated with the resource `res_gpu2` and keep a handle to the corresponding event object by assigning it to a local variable `e`:

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

We require that the next kernel we launch to wait for the kernel launched on the stream associated with the resource `res_gpu2` to complete. Therefore, we enqueue a wait on that event on the `res_gpu1` resource:

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

Now that the second GPU kernel is complete, we launch a second kernel on the stream associated with the resource `res_gpu1`:

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

Next, we enqueue a memcpy operation on the resource `res_gpu1` to copy the GPU array `d_array` to the host array `h_array`:

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

Lastly, we use the copied data in a kernel executed on the host:

```
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;}
  }
);
```

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 written using RAJA local arrays in a `RAJA::kernel_param` kernel. We show this next, 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>
                  >
                >
              >
            >;


//
// Define the kernel
//
```

(continues on next page)

(continued from previous page)

```
RAJA::kernel_param<POL> ( RAJA::make_tuple(RAJA::TypedRangeSegment<int>(0,5),
                                           RAJA::TypedRangeSegment<int<(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 striding order (see *View and Layout* for details)
- Array dimensions

The local array instances are passed to the kernel in a tuple after the iteration space tuple.

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' statements, which allocates the memory for the local arrays when the kernel executes. The `InitLocalMem` statement type has two parameters. One for the memory type `RAJA::cpu_tile_mem`, and one for specifying which parameter tuple entries correspond to the local arrays `RAJA::ParamList<0, 1>`. The local array initialization is done in the first lambda expression, and the local array values are printed in the second lambda expression.

Note: `RAJA::LocalArray` types support arbitrary dimensions and extents in each dimension.

Memory Policies

`RAJA::LocalArray` supports CPU stack-allocated memory and CUDA or HIP 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 loops. Typical loop tiling involves partitioning an iteration space into a collection of "tiles" and then iterating over tiles in outer loops and indices 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, consider an operation performed using a C-style for-loop with a range of [0, 10):

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

This May be written 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 loop tiling can be written using RAJA with variations that use different `RAJA::kernel` execution policy statement types.

Here is a way to write the tiled loop kernel above using `RAJA::kernel`:

```
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::TypedRangeSegment<int>(0,10)),
  [=] (int i) {
    // kernel body using index 'i'
  }
);
```

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

Note: When using `RAJA::statement::Tile` and `RAJA::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 iteration space in the space tuple passed to the `RAJA::kernel` method.

RAJA also provides alternative 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::TypedRangeSegment<int>(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 `RAJA::statement::TileTCount` type indicates that the tile number will be passed to the lambda expression and the `RAJA::statement::ForICount` type indicates that the local tile loop index will be passed to the lambda expression. Storage for these values is specified in the parameter tuple, the second argument passed to the `RAJA::kernel_param` method. The `RAJA::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 `RAJA::Param` statement types given for the loop statement types follow.

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
    - a work dispatch 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, *Multi-dimensional loops using simple loop APIs (RAJA::CombiningAdapter)*.

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

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 four components, a work execution policy, a work ordering policy, a work storage policy, and a work dispatch 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,
                             RAJA::indirect_function_call_dispatch >;
```

is a workgroup policy that will run loops sequentially on the host in the order they were enqueued, stores the loop bodies sequentially in single buffer in memory, and dispatches each loop using a function pointer.

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.

Work Ordering Policies	Brief description
ordered	Execute loops sequentially in the order they were enqueued using forall.
reverse_ordered	Execute loops sequentially in the reverse of the order order they were enqueued using forall.
unordered_cuda_loops_y_block_iter_x_threadblock_average	Execute loops in parallel by mapping each loop to a set of cuda blocks with the y block index in the y direction and a cuda kernel. Each loop is given a number of threads over one of more blocks in the x direction equal to the average number of iterations of all the loops rounded up to a multiple of the block size.

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	Store 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.

The work dispatch policy determines the technique used to dispatch from type erased storage to the loops or iterations of each range and loop body pair.

Work Dispatch Policies	Brief description
indirect_function_call_dispatch	Dispatch using function pointers.
indirect_virtual_function_dispatch	Dispatch using virtual functions in a class hierarchy.
direct_dispatch<	Dispatch using a switch statement like
camp::list<Range, Callable>...>	coding to pick the right pair of Range and Callable types from the template parameter pack. You may only enqueue a range and callable pair that is in the list of types in the policy.

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 *Multi-dimensional 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 >;
```

(continues on next page)

(continued from previous page)

```
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 and Examples*. 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, but it is ready for interested users to try.

- We provide a basic description here so that interested users can take a look, try it out, and provide input if they wish to do so. The RAJA team values early feedback from users on new capabilities.
- There are no usage examples available in RAJA yet, except for tests. Examples will be made available as they are developed.

The aim of the RAJA API for SIMD/SIMT programming described in this section is to make an implementation perform as well as if one used SIMD/SIMT intrinsics directly in her code, but without the software complexity and maintenance burden associated with doing that. In particular, we want to *guarantee* that specified vectorization occurs without requiring users to manually insert intrinsics in their code or rely on compiler auto-vectorization implementations.

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

Currently, the main abstractions in RAJA for SIMD/SIMT programming are:

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

Using these abstractions, RAJA provides an expression-template system that allows users to write linear algebra expressions on arbitrarily sized scalars, vectors, and matrices and have the appropriate SIMD/SIMT instructions performed during expression evaluation. These capabilities integrate with RAJA *View and Layout* capabilities, which insulate load/store and other operations from user code.

Why Are We Doing This?

Quoting Tim Foley in Matt Pharr's blog – "Auto-vectorization is not a programming model". This is true, of course, unless you consider "hope for the best" that the compiler optimizes the way you want to be a sound code development strategy.

Compiler auto-vectorization is problematic for multiple reasons. First, when vectorization is not explicit in source code, compilers must divine correctness when attempting to apply vectorization optimizations. Most compilers are very conservative in this regard, due to the possibility of data aliasing in C and C++ and prioritizing correctness over performance. Thus, many vectorization opportunities are usually missed when one relies solely on compiler auto-vectorization. Second, every compiler will treat your code differently since compiler implementations use different optimization heuristics, even in different versions of the same compiler. So performance portability is not just an issue with respect to hardware, but also for compilers. Third, it is generally impossible for most application developers to clearly understand the choices made by compilers during optimization processes.

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.

Writing GPU code makes a programmer 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 some convergence between SIMD and GPU programming by providing uniform access to hardware-specific acceleration.

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

---

Register

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

`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 and HIP wavefront for NVIDIA and AMD GPUs, respectively. Scalar support is provided for all hardware for portability and experimentation/analysis. Extensions to support other architectures may be forthcoming as they are needed and requested by users.

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

Register Operations

`RAJA::expt::Register` provides various operations which include:

- 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 code example shows how to use the `RAJA::expt::Register` class to perform a DAXPY kernel with AVX2 SIMD instructions. While we do not recommend that you write code directly using the Register class, but instead use the higher level VectorRegister abstraction, we use the Register type here to illustrate the basics mechanics of SIMD vectorization:

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

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

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

// Compute daxpy in chunks of 4 values (register width) at a 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 and store in a register
  reg_t z = a * x + y;

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

// Loop postamble code to complete daxpy operation when array length
// is not an integer multiple of the register width
int remainder = len % reg_width;
if (remainder) {
  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 and 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 intrinsics into the operation calls. Since `RAJA::expt::Register` provides overloads of basic arithmetic operations, the SIMD DAXPY operation `z = a * x + y` looks like vanilla scalar code.

Because 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 explicit *postamble* code to handle cases where the array length `len` is not an integer multiple of the register width `reg_width`. The postamble code performs the DAXPY operation on the *remainder* of the array that is excluded from the for-loop, which is strided by the register width.

The need to write extra postamble code should make clear one reason why we do not recommend using ‘‘RAJA::Register‘‘ directly in application code.

Vector Register

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

`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: data type `T`, vector register policy `REGISTER_POLICY`, and `NUM_ELEM` which is the number of data elements of type `T` that fit in a register. The last two of these template parameters have defaults for all cases, so a user need note provide them in most cases.

Recall that we said earlier that we do not recommended using `RAJA::expt::Register` directly. One important reason for this is that decoupling the vector length from hardware register size allows one to write simpler, more readable code that is easier to get correct. This should be clear from the code example below, when compared to the previous code example.

Vector Register DAXPY Example

The following code example shows the DAXPY computation discussed above, but written using `RAJA::expt::VectorRegister`, `RAJA::expt::VectorIndex`, and `RAJA::View` types. Using these types, we can write cleaner, more concise code that is easier to get correct because it is simpler. For example, we do not have to write the postamble 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 );

// The 'all' variable gets the length of the arrays from the vX, vY, and
// vZ View objects and encodes the vector register 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
// in the executable
vZ( all ) = a * vX( all ) + vY( all );
```

It should be clear that this code has several advantages over the previous code example. It is guaranteed to vectorize as before, but it is much easier to read, get correct, and maintain since the `RAJA::View` class handles the looping and postamble code automatically for arrays of arbitrary size. The `RAJA::View` class provides overloads of the arithmetic operations based on the `all` variable and inserts the appropriate SIMD instructions and load/store operations to vectorize the operations that were explicit 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 is shown here.

Expression Templates

The figure below shows the sequence of SIMD operations, as they are parsed to form of an *abstract syntax tree (AST)*, for the DAXPY code in the vector register code example above.

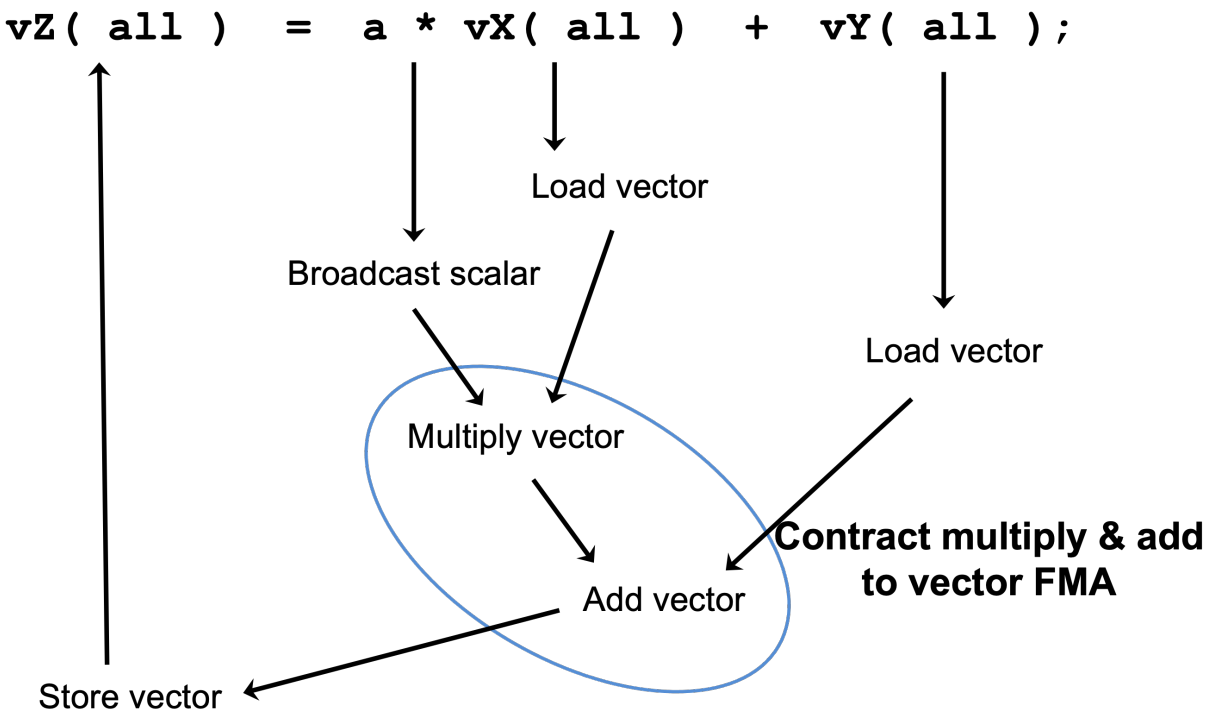

Fig. 1: An AST illustration of the SIMD operations in the DAXPY code.

During compilation, a tree of *expression template* objects is constructed based on the order of operations that appear in the DAXPY 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 *Load/Store* 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 associated with the view `vZ` on the left-hand side of the equal sign.

CPU/GPU Portability

It is important to note that the code in the example above can only run on a CPU; i.e., it is *not* portable to run on either a CPU or 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 = ...;

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 a CPU and an CUDA GPU based on the parameters in the `pol_t` loop policy. The CPU version will be the same as the version described earlier. 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.

Tensor Register

`RAJA::expt::TensorRegister< >` is a class template that provides a higher-level interface on top of `RAJA::expt::Register`. `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 higher-level abstraction types that RAJA provides and which are described below.

Matrix Registers

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

- `RAJA::expt::SquareMatrixRegister<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 in the `RAJA::expt::VectorRegister` example.

Matrix operations support matrix-matrix, matrix-vector, 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 GPU warp/wavefront registers, which are 32-wide for CUDA and 64-wide for HIP.

Here is a code example that performs the matrix-analogue of the vector DAXPY operation 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 );

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 on two-dimensional matrices. The kernel code is easy to read, it is guaranteed to vectorize, and iterating over the data

is handled by RAJA view objects (register-width sized chunk, plus postamble scalar operations), and it can run on a CPU or NVIDIA GPU. As before, the `RAJA::View` arithmetic operation overloads insert the appropriate vector instructions in the code.

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.

Plugins Quick Start

Static Plugins

1. Build RAJA normally.
2. Use an `#include` statement in your code or pass options to the compiler to load your plugin file with your project at compile time. For example: `g++ project.cpp plugin.cpp -lRAJA -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 `.so` extension. For example: `g++ plugin.cpp -lRAJA -fPIC -shared -o plugin.so`.
3. Set the environment variable `RAJA_PLUGINS` to 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 a directory, all `.so` files in that directory will be loaded.
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 methods to call in your code: `init_plugins` and `finalize_plugins`. These will call the corresponding `init` and `finalize` methods, respectively, of *every* currently loaded plugin. It's worth noting that plugins don't require either an init or finalize method by default.

- `RAJA::util::init_plugins();` will call the `init` method of every currently loaded plugin.
- `RAJA::util::init_plugins("path/to/plugins");` will call the `init` method of every currently loaded plugin and, in addition, will also dynamically load plugins located at the given path.
- `RAJA::util::finalize_plugins();` will call the `finalize` method of every currently loaded plugin.

Creating Plugins For RAJA

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

Plugin API methods

The following list summarizes the virtual methods in the `RAJA::util::PluginStrategy` base class.

- `void init(const PluginOptions& p) override {}` is called on all plugins when a user calls `init_plugins()`
- `void preCapture(const PluginContext& p) override {}` is called before lambda capture in RAJA kernel execution methods.
- `void postCapture(const PluginContext& p) override {}` is called after lambda capture in RAJA kernel execution methods.
- `void preLaunch(const PluginContext& p) override {}` is called before a RAJA kernel execution method runs a kernel.
- `void postLaunch(const PluginContext& p) override {}` is called after a RAJA kernel execution method runs a kernel.
- `void finalize() override {}` is called on all plugins when a user calls `finalize_plugins`. This will also unload all currently loaded plugins.

Note: The pre/post methods above are automatically called before and after executing a kernel with `RAJA::forall` or `RAJA::kernel` kernel execution methods.

Note: The `init` and `finalize` methods are never called 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, it must be loaded with either an `#include` statement in the project source code or by calling the following method in the project source code, which adds the plugin to the RAJA ‘‘PluginRegistry‘::‘

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

In either case, the plugin will be loaded every time the compiled project executable is run.

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 method that returns a pointer to an instance of your plugin:

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

   Note that using `extern "C"` is required to search for the `getPlugin()` method call for the dynamically loaded plugin correctly.

2. The plugin must be compiled to be a shared object with a `.so` extension. For example: `g++ plugin.cpp -lRAJA -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` is a standard flag for linking 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 methods flagged with `extern "C"`.

3. The `RAJA_PLUGINS` environment variable must be set, or the project code must call `RAJA::util::init_plugins("path");`. Either of these approaches is required to supply the path to either a directory containing the plugin or its `.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
    {
```

(continues on next page)

(continued from previous page)

```
      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. One option for managing such data is to use CHAI, which 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 determined by the RAJA execution 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 CHAI 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;
});
```

(continues on next page)

(continued from previous page)

```
RAJA::forall<RAJA::loop_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 memory to GPU memory, making it available for access in the RAJA kernel. The data 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.

6.1.5 Application Considerations

Warning: Comming soon!! Stay tuned.

6.1.6 RAJA Tutorial and Examples

The following sections contain tutorial material and examples that describe how to use RAJA features.

RAJA Tutorial

This section contains a self-paced tutorial that shows how to use many RAJA features by way of a sequence of examples and exercises. Each exercise is located in files in the `RAJA/exercises` directory, one *exercise* file with code sections removed and comments containing instructions to fill in the missing code parts and one *solution* file containing complete working code to compare with and for guidance if you get stuck working on the exercise file. You are encouraged to build and run the exercises and modify them to try out different variations.

We also maintain a repository of tutorial slide presentations RAJA Tutorials Repo which we use when we give in-person or virtual online tutorials in various venues. The presentations complement the material found here. The tutorial material evolves as we add new features to RAJA, so refer to it periodically if you are interested in learning about new things in RAJA.

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.

It is important to note that RAJA does not provide a memory model. This is by design as application developers who use RAJA prefer to manage memory in different ways. 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. The RAJA Portability Suite contains other libraries, namely CHAI and Umpire, that complement RAJA by providing alternatives to manual programming model specific memory operations.

Note: Most of the CUDA GPU exercises use unified memory (UM) via a simple memory manager capability provided in a file in the `RAJA/exercises` directory. HIP GPU exercises use explicit host and device memory allocations and explicit memory copy operations to move data between the two.

A Little C++ Background

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 use RAJA successfully.

RAJA is almost an entirely header-only library that makes heavy use of C++ templates. 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 type-generic code and have the compiler generate an implementation for each set of template parameter types specified. For example, the `RAJA::forall` method to execute loop kernels is essentially method defined as:

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

Here, "ExecPol", "IdxType", and "LoopBody" are C++ types that a user specifies in her code and which are seen by the compiler when the code is built. For example:

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

is a sequential CPU RAJA kernel that performs an element-by-element vector sum. The C-style analogue of this kernel is:

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

The execution policy type `RAJA::loop_exec` template argument is used to choose as specific implementation of the `RAJA::forall` method. The `IdxType` and `LoopBody` types are deduced by the compiler based the arguments passed to the `RAJA::forall` method; i.e., the `IdxType` is the stride-1 index range:

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

and the `LoopBody` type is 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 kernel execution templates, such as `RAJA::forall` and `RAJA::kernel` that we will describe in detail later, pass arguments to lambdas based on usage and context such as 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 included in the capture list with no extra symbols are captured by value.** Variables captured by value are effectively *const* inside the lambda expression body and cannot be written to. Capture-by-reference is accomplished by using the reference symbol '&' before the variable name similar to C++ method arguments. 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.
```

Note: A variable that is captured by value in a lambda expression is **read-only.**

A Few Notes About Lambda Usage With RAJA

There are several issues to note about using C++ lambda expressions to represent kernel bodies with RAJA. We describe them here.

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

  We recommend *capture by-value* for all lambda kernel 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.

- **The '__device__' annotation is required for device execution using CUDA or HIP.**

  Any lambda passed to a CUDA or HIP execution context (or function called from a 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 in 'device' code.

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

- **Use 'host-device' annotation on a lambda carefully.**

  RAJA provides the macro `RAJA_HOST_DEVICE` to support the dual annotation `__host__ __device__`, which makes a lambda or function callable from CPU or GPU 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**. Although compiler improvements in recent years have significantly improved support for host-device lambda expressions, 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 compiler (e.g., nvcc) you are using. To be sure that your code does not suffer in performance, we recommend comparing CPU execution timings of important kernels with and without the `__host__ __device__` annotation.

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

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

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

  This fact is due to the C++ standard. If you need 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 device 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 was resolved in the 10.1 release of CUDA. If you are using an earlier version, an implementation similar to the one above will be required.

RAJA Examples and Exercises

The remainder of this tutorial illustrates how to use RAJA features with working code examples and interactive exercises. Files containing the exercise source code are located in the `RAJA/exercises` directory. Additional information about the RAJA features used can be found in *RAJA Features*.

The examples demonstrate CPU execution (sequential and OpenMP multithreading) and GPU execution (CUDA and/or HIP). Examples that show how to use RAJA with other parallel programming model back-ends will appear in future RAJA releases. For adventurous users who wish to try experimental RAJA back-end support, 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*.

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, sorts, and RAJA data views.

Basic Loop Execution: Vector Addition

This section contains an exercise file `RAJA/exercises/vector-addition.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/vector-addition_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make vector-addition` and `make vector-addition_solution` from the build directory.

Key RAJA features shown in this example are:

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

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_ref[i] = a[i] + b[i];
}
```

RAJA Variants

For the RAJA variants of the vector addition kernel, 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 the object:

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

for the iteration space, which is contiguous sequence of integral values [0, N) (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 a RAJA sequential variant, we use the `RAJA::seq_exec` execution policy type:

```
  RAJA::forall< RAJA::seq_exec >(
    RAJA::TypedRangeSegment<int>(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
```

An alternative RAJA policy is:

```
RAJA::loop_exec
```

which allows the compiler to generate optimizations based on how its internal 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::TypedRangeSegment<int>(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. In particular, this is what you would get if you wrote the kernel using a C-style loop with an OpenMP pragma directly:

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

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::TypedRangeSegment<int>(0, N),
↪
    [=] RAJA_DEVICE (int i) {
    d_c[i] = d_a[i] + d_b[i];
  });
```

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

Note that the CUDA execution policy type requires a template argument `CUDA_BLOCK_SIZE`, which specifies the number of threads to run in each CUDA thread block launched to run the kernel.

For additional performance tuning options, the `RAJA::cuda_exec_explicit` policy is also provided, which allows a user to specify the minimum number of thread blocks to launch at a time on each streaming multiprocessor (SM):

```
  const bool Asynchronous = true;

  RAJA::forall<RAJA::cuda_exec_explicit<CUDA_BLOCK_SIZE, 2, Asynchronous>>
↪(RAJA::TypedRangeSegment<int>(0, N),
    [=] RAJA_DEVICE (int i) {
    d_c[i] = d_a[i] + d_b[i];
  });
```

Note that the third boolean template argument is used to express whether the kernel launch is synchronous or asynchronous. This is optional and is 'false' by default. A similar defaulted optional argument is supported for other RAJA GPU (e.g., CUDA or HIP) policies.

Lastly, 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::TypedRangeSegment<int>(0, N),
    [=] RAJA_DEVICE (int i) {
    d_c[i] = d_a[i] + d_b[i];
  });
```

Iteration Spaces: Segments and IndexSets

This section contains an exercise file `RAJA/exercises/segment-indexset-basics.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/segment-indexset-basics_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make segment-indexset-basics` and `make segment-indexset-basics_solution` from the build directory.

Key RAJA features shown in this example are:

- `RAJA::forall` loop execution template
- `RAJA::TypedRangeSegment`, `RAJA::TypedRangeStrideSegment`, and `RAJA::TypedListSegment` iteration space constructs
- `RAJA::TypedIndexSet` container and associated execution policies

The concepts of iteration spaces and associated Loop variables are central to writing kernels in RAJA. RAJA provides 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.

The examples in this section focus on how to use RAJA index sets and iteration space segments, such as index ranges and lists of indices. Lists of indices are important for 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 help improve 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.

Although the constructs described in the section are useful in numerical computations and parallel execution, the examples only contain print statements and sequential execution. The goal is to show you how to use RAJA iteration space constructs.

RAJA Segments

A RAJA *Segment* represents a set of indices that one wants to execute as a unit for a kernel. RAJA provides the following Segment types:

- `RAJA::TypedRangeSegment` represents a stride-1 range
- `RAJA::TypedRangeStrideSegment` represents a (non-unit) stride range
- `RAJA::TypedListSegment` represents an arbitrary set of indices

These segment types are used in `RAJA::forall` and other RAJA kernel execution mechanisms to define the iteration space for a kernel.

After we briefly introduce these types, we will present several examples using them.

TypedRangeSegment

A `RAJA::TypedRangeSegment` is the fundamental type for defining a stride-1 (i.e., contiguous) range of indices. This is illustrated in the figure below.

Fig. 2: A range segment defines a stride-1 index range [beg, end).

One creates a range segment object as follows:

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

Any integral type can be given as the template parameter.

Note: When using a RAJA range segment, no loop iterations will be run when begin >= end.

TypedRangeStrideSegment

A `RAJA::TypedRangeStrideSegment` defines a range with a constant stride, including negative stride values if needed. This is illustrated in the figure below.

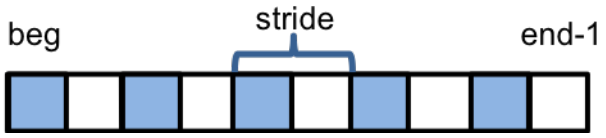

Fig. 3: A range-stride segment defines an index range with arbitrary stride [beg, end, stride). In the figure the stride is 2.

One creates a range stride segment object as follows:

```
// A stride-3 index range [beg, end) using type int.
RAJA::TypedRangeStrideSegment<int> my_stride2_range(beg, end, 3);

// A index range with -1 stride [0, N-1) using type int
RAJA::TypedRangeStrideSegment<int> my_neg1_range( N-1, -1, -1);
```

Any integral type can be given as the template parameter.

When the negative-stride segment above is passed to a `RAJA::forall` method, for example, the loop will run in reverse order with iterates:

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

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

TypedListSegment

A `RAJA::TypedListSegment` is used to define an arbitrary set of indices, akin to an indirection array. This is illustrated in the figure below.

Fig. 4: A list segment defines an arbitrary collection of indices. Here, we have a list segment with 5 irregularly-spaced indices.

One creates a list segment object by passing a container 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 indices where the indices are
// stored in the CUDA device memory space
camp::resources::Resource cuda_res{camp::resources::Cuda()};
RAJA::TypedListSegment<int> idx_list( idx[0], cuda_res );

// Alternatively
RAJA::TypedListSegment<int> idx_list( &idx[0], idx.size(),
                                      cuda_res );
```

When the list segment above is passed to a `RAJA::forall` method, for example, the kernel will execute with iterates:

```
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. If the indices are in a container, such as `std::vector` that provides `begin()`, `end()`, and `size()` methods, the container can be passed to the constructor and the length argument is not required.

Note: Currently, a camp resource object must be passed to a list segment constructor to copy the indices in the indices into the proper memory space for a kernel to execute (as shown above). In the future, this will change and the user will be responsible for providing the indices in the proper memory space.

IndexSets

A `RAJA::TypedIndexSet` is a container that can hold an arbitrary collection of segment objects. The following figure shows an index set with two contiguous ranges and an irregularly-spaced list of indices.

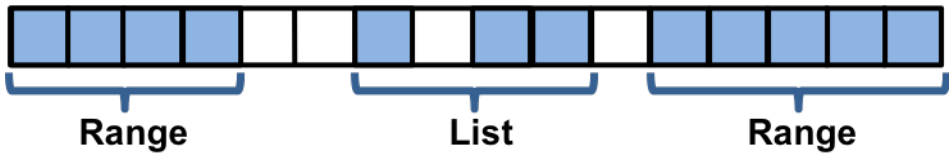

Fig. 5: An index set with two range segments and one list segment.

We can create such an index set as follows:

```
// Create an index set that can hold range and list segments with
// int index value type
RAJA::TypedIndexSet< RAJA::TypedRangeSegment<int>,
                     RAJA::TypedListSegment<int> > iset;

// Add two range segments and one list segment to the index set
iset.push_back( RAJA::TypedRangeSegment<int>( ... ) );
iset.push_back( RAJA::TypedListSegment<int>(...) );
iset.push_back( RAJA::TypedRangeSegment<int>( ... ) );
```

A `RAJA::TypedIndexSet` object can be passed to a RAJA kernel execution method, such as `RAJA::forall` to execute all segments in the index set with one method call. We will show this in detail in the examples below.

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.

Segment and IndexSet Examples

The examples in this section illustrate how the segment types that RAJA provides can be used to define kernel iteration spaces. We use the following type aliases to make the code more compact:

```
using IdxType = int;
using RangeSegType = RAJA::TypedRangeSegment<IdxType>;
using RangeStrideSegType = RAJA::TypedRangeStrideSegment<IdxType>;
using ListSegType = RAJA::TypedListSegment<IdxType>;
using IndexSetType = RAJA::TypedIndexSet< RangeSegType, ListSegType >;
```

Stride-1 Indexing

Consider a simple C-style kernel that prints a contiguous sequence of values:

```
for (IdxType i = 0; i < 20; i++) {
  std::cout << i << "  ";
}
```

When run, the kernel prints the following sequence, as expected:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Three RAJA variants of the kernel using a `RAJA::TypedRangeSegment`, a `RAJA::TypedRangeStrideSegment`, and a `RAJA::TypedListSegment` are:

```
RAJA::forall<RAJA::seq_exec>(RangeSegType(0, 20), [=] (IdxType i) {
  std::cout << i << "  ";
});
```

```
RAJA::forall<RAJA::seq_exec>(RangeStrideSegType(0, 20, 1), [=] (IdxType i) {
  std::cout << i << "  ";
});
```

```
//
// Collect indices in a vector to create list segment
//
std::vector<IdxType> idx;
for (IdxType i = 0; i < 20; ++i) {
  idx.push_back(i);
}

ListSegType idx_list1( idx, host_res );

RAJA::forall<RAJA::seq_exec>(idx_list1, [=] (IdxType i) {
  std::cout << i << "  ";
});
```

Each of these variants prints the same integer sequence shown above.

One interesting thing to note is that with `RAJA::TypedListSegment` and `RAJA::forall`, the actual iteration value is passed to the lambda loop body. So the indirection array concept is not visible. In contrast, in C-style code, one has to manually retrieve the index value from the indirection array to achieve the desired result. For example:

```
IdxType iis = static_cast<IdxType>(idx.size());  // to avoid compiler warning
for (IdxType ii = 0; ii < iis; ++ii) {
  std::cout << idx[ ii ] << "  ";
}
```

Non-unit Stride Indexing

Consider the following C-style kernel that prints the integer sequence discussed earlier in reverse order:

```
for (IdxType i = 19; i > -1; i--) {
  std::cout << i << "  ";
}
```

We can accomplish the same result using a `RAJA::TypedRangeStrideSegment`:

```
RAJA::forall<RAJA::seq_exec>(RangeStrideSegType(19, -1, -1), [=] (IdxType i) {
  std::cout << i << "  ";
});
```

Alternatively, we can use a `RAJA::TypedListSegment`, where we reverse the index array we used earlier to define the appropriate list segment:

```
//
// Reverse the order of indices in the vector
//
std::reverse( idx.begin(), idx.end() );
ListSegType idx_list1_reverse( &idx[0], idx.size(), host_res );

RAJA::forall<RAJA::seq_exec>(idx_list1_reverse, [=] (IdxType i) {
  std::cout << i << "  ";
});
```

The more common use of the `RAJA::TypedRangeStrideSegment` type is to run constant strided loops with a positive non-unit stride. For example:

```
RAJA::forall<RAJA::seq_exec>(RangeStrideSegType(0, 20, 2), [=] (IdxType i) {
  std::cout << i << "  ";
});
```

The C-style equivalent of this is:

```
for (IdxType i = 0; i < 20; i += 2) {
  std::cout << i << "  ";
}
```

IndexSets: Complex Iteration Spaces

We noted earlier that `RAJA::TypedIndexSet` objects can be used to partition iteration spaces into disjoint parts. Among other things, this can be useful to expose parallelism in algorithms that would otherwise require significant code transformation to do so. Please see *Iteration Space Coloring: Mesh Vertex Sum* for discussion of an example that illustrates this.

Here is an example that uses two `RAJA::TypedRangeSegment` objects in an index set to represent an iteration space broken into two disjoint contiguous intervals:

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

IndexSetType is2;
is2.push_back( RangeSegType(0, 10) );
is2.push_back( RangeSegType(15, 20) );

RAJA::forall<SEQ_ISET_EXECPOL>(is2, [=] (IdxType i) {
  std::cout << i << "  ";
});
```

The integer sequence that is printed is:

```
0  1  2  3  4  5  6  7  8  9  15  16  17  18  19
```

as we expect.

The execution policy type when using a RAJA index set is a *two-level* policy. The first level specifies how to iterate over the segments in the index set, such as sequentially or in parallel using OpenMP. The second level is the execution policy used to execute each segment.

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 segments. The second parameter specifies how the kernel will execute each segment over each segment. See *RAJA IndexSet Execution Policies* for more information about RAJA index set execution policies.

It is worth noting that a C-style version of this kernel requires either an indirection array to run in one loop or two for-loops. For example:

```
for (IdxType i = 0; i < 10; ++i) {
  std::cout << i << "  ";
}
for (IdxType i = 15; i < 20; ++i) {
  std::cout << i << "  ";
}
```

Finally, we show an example that uses an index set holding two range segments and one list segment to partition an iteration space into three parts:

```
IndexSetType is3;

is3.push_back( RangeSegType(0, 8) );

IdxType indx[ ] = {10, 11, 14, 20, 22};
ListSegType list2( indx, 5, host_res );
is3.push_back( list2 );

is3.push_back( RangeSegType(24, 28) );

RAJA::forall<SEQ_ISET_EXECPOL>(is3, [=] (IdxType i) {
  std::cout << i << "  ";
});
```

The integer sequence that is printed is:

```
0  1  2  3  4  5  6  7  10  11  14  20  22  24  25  26  27
```

Iteration Space Coloring: Mesh Vertex Sum

This section contains an exercise file `RAJA/exercises/vertexsum-indexset.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/vertexsum-indexset_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make vertexsum-indexset` and `make vertexsum-indexset_solution` from the build directory.

Key RAJA features shown in this example are:

- `RAJA::forall` loop execution template method

- `RAJA::TypedListSegment` iteration space construct
- `RAJA::TypedIndexSet` 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.

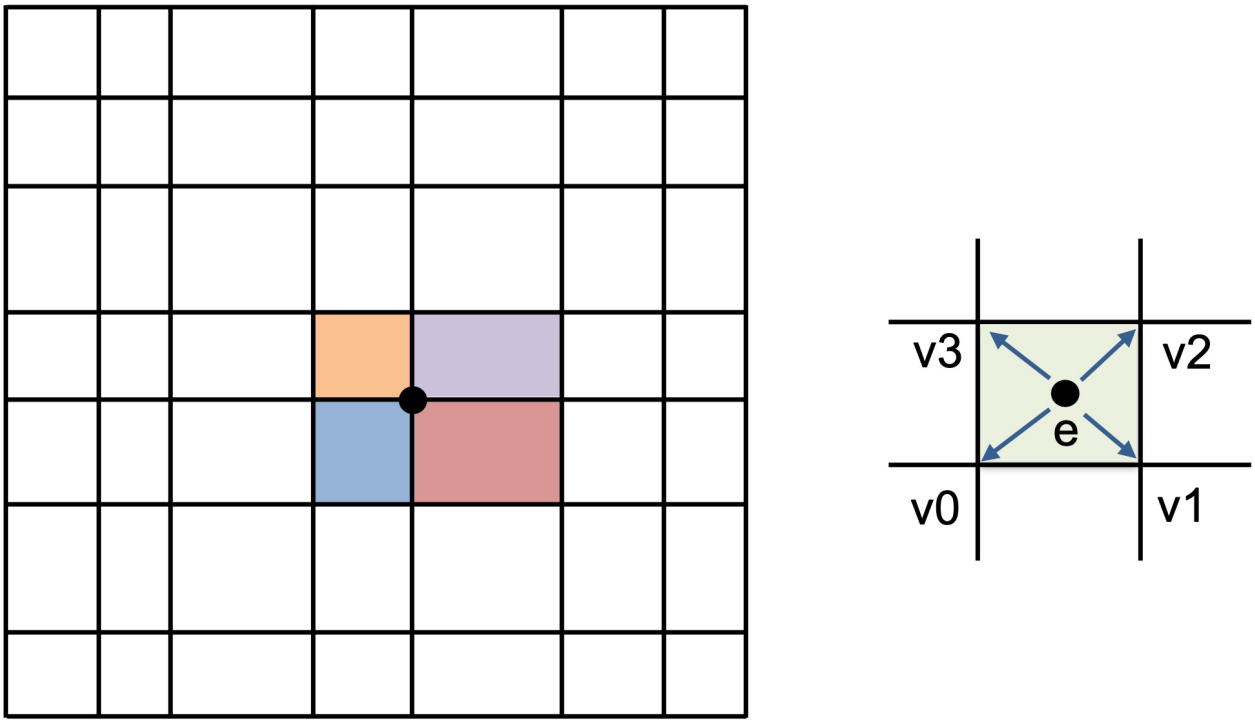

Fig. 6: The "area" of each vertex is the sum of an area contribution from each element sharing the vertex (left). In particular, one quarter of the area of each mesh element is summed to the vertices surrounding the element (right).

Each sum is an average of the area of the four mesh elements that share the vertex. In many "staggered mesh" applications, an operation like this 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.

We start by setting the size of the mesh, specifically, the total number of elements and vertices and the number of elements and vertices in each direction:

```
//
// 2D mesh has N^2 elements (N+1)^2 vertices.
//
  constexpr int N = 1000;
  constexpr int Nelem = N;
  constexpr int Nelem_tot = Nelem * Nelem;
  constexpr int Nvert = N + 1;
  constexpr int Nvert_tot = Nvert * Nvert;
```

We also set up an array to map each element to its four surrounding vertices and set the area of each element:

```
//
// Define mesh spacing factor 'h' and set up elem to vertex mapping array.
//
```

(continues on next page)

(continued from previous page)

```
  constexpr double h = 0.1;

  for (int ie = 0; ie < Nelem_tot; ++ie) {
    int j = ie / Nelem;
    int imap = 4 * ie ;
    e2v_map[imap] = ie + j;
    e2v_map[imap+1] = ie + j + 1;
    e2v_map[imap+2] = ie + j + Nvert;
    e2v_map[imap+3] = ie + j + 1 + Nvert;
  }

//
// Initialize element areas so each element area
// depends on the i,j coordinates of the element.
//
  std::memset(areae, 0, Nelem_tot * sizeof(double));

  for (int ie = 0; ie < Nelem_tot; ++ie) {
    int i = ie % Nelem;
    int j = ie / Nelem;
    areae[ie] = h*(i+1) * h*(j+1);
  }
```

Then, a sequential C-style version of the vertex area calculation looks like this:

```
  std::memset(areav_ref, 0, Nvert_tot * sizeof(double));

  for (int ie = 0; ie < Nelem_tot; ++ie) {
    int* iv = &(e2v_map[4*ie]);
    areav_ref[ iv[0] ] += areae[ie] / 4.0 ;
    areav_ref[ iv[1] ] += areae[ie] / 4.0 ;
    areav_ref[ iv[2] ] += areae[ie] / 4.0 ;
    areav_ref[ iv[3] ] += areae[ie] / 4.0 ;
  }
```

We can't parallelize the entire computation at once due to potential race conditions where multiple threads may attempt to sum to a shared element vertex simultaneously. However, we can parallelize the computation in parts. Here is a C-style OpenMP parallel implementation:

```
  std::memset(areav, 0, Nvert_tot * sizeof(double));

  for (int icol = 0; icol < 4; ++icol) {
     const std::vector<int>& ievec = idx[icol];
     const int len = static_cast<int>(ievec.size());

     #pragma omp parallel for
     for (int i = 0; i < len; ++i) {
        int ie = ievec[i];
        int* iv = &(e2v_map[4*ie]);
        areav[ iv[0] ] += areae[ie] / 4.0 ;
        areav[ iv[1] ] += areae[ie] / 4.0 ;
        areav[ iv[2] ] += areae[ie] / 4.0 ;
        areav[ iv[3] ] += areae[ie] / 4.0 ;
     }

  }
```

What we've done is broken up the computation into four parts, each of which can safely run in parallel because there are no overlapping writes to the same entry in the vertex area array in each parallel section. Note that there is an outer loop on length four, one iteration for each of the elements that share a vertex. Inside the loop, we iterate over a subset of elements in parallel using an indexing area that guarantees that we will have no data races. In other words, we have "colored" the elements as shown in the figure below.

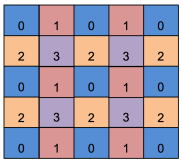

Fig. 7: We partition the mesh elements into four disjoint subsets shown by the colors and numbers so that within each subset no two elements share a vertex.

For completeness, the computation of the four element indexing arrays is:

```
//
// Gather the element indices for each color in a vector.
//
  std::vector< std::vector<int> > idx(4);

  for (int ie = 0; ie < Nelem_tot; ++ie) {
    int i = ie % Nelem;
    int j = ie / Nelem;
    if ( i % 2 == 0 ) {
      if ( j % 2 == 0 ) {
        idx[0].push_back(ie);
      } else {
        idx[2].push_back(ie);
      }
    } else {
      if ( j % 2 == 0 ) {
        idx[1].push_back(ie);
      } else {
        idx[3].push_back(ie);
      }
    }
  }
```

RAJA Parallel Variants

To implement the vertex sum calculation using RAJA, we employ `RAJA::TypedListSegment` iteration space objects to enumerate the mesh elements for each color and put them in a `RAJA::TypedIndexSet` object. This allows us to execute the entire calculation using one `RAJA::forall` call.

We declare a type alias for the list segments to make the code more compact:

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

Then, we build the index set:

```
  RAJA::TypedIndexSet<SegmentType> colorset;

  colorset.push_back( SegmentType(&idx[0][0], idx[0].size(), host_res) );
```

(continues on next page)

(continued from previous page)

```
colorset.push_back( SegmentType(&idx[1][0], idx[1].size(), host_res) );
colorset.push_back( SegmentType(&idx[2][0], idx[2].size(), host_res) );
colorset.push_back( SegmentType(&idx[3][0], idx[3].size(), host_res) );
```

Note that we construct the list segments using the arrays we made earlier to partition the elements. Then, we push them onto the index set.

Now, we can use a two-level index set execution policy that iterates over the segments sequentially and executes each segment in parallel using OpenMP multithreading to run the kernel:

```
using EXEC_POL1 = RAJA::ExecPolicy<RAJA::seq_segit,
                                   RAJA::omp_parallel_for_exec>;

RAJA::forall<EXEC_POL1>(colorset, [=](int ie) {
  int* iv = &(e2v_map[4*ie]);
  areav[ iv[0] ] += areae[ie] / 4.0 ;
  areav[ iv[1] ] += areae[ie] / 4.0 ;
  areav[ iv[2] ] += areae[ie] / 4.0 ;
  areav[ iv[3] ] += areae[ie] / 4.0 ;
});
```

The execution of the RAJA version is similar to the C-style OpenMP variant shown earlier, where we executed four OpenMP parallel loops in sequence, but the code is more concise. In particular, we execute four parallel OpenMP loops, one for each list segment in the index set. Also, note that we do not have to manually extract the element index from the segments like we did earlier since RAJA passes the segment entries directly to the lambda expression.

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

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

RAJA::forall<EXEC_POL2>(cuda_colorset, [=] RAJA_DEVICE (int ie) {
  int* iv = &(e2v_map[4*ie]);
  areav[ iv[0] ] += areae[ie] / 4.0 ;
  areav[ iv[1] ] += areae[ie] / 4.0 ;
  areav[ iv[2] ] += areae[ie] / 4.0 ;
  areav[ iv[3] ] += areae[ie] / 4.0 ;
});
```

The only differences here are that we have marked the lambda loop body with the `RAJA_DEVICE` macro, used a CUDA segment execution policy, and built a new index set with list segments created using a CUDA resource so that the indices live in device memory.

The RAJA HIP variant, which we show for completeness, is similar:

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

RAJA::forall<EXEC_POL3>(hip_colorset, [=] RAJA_DEVICE (int ie) {
  int* iv = &(d_e2v_map[4*ie]);
  d_areav[ iv[0] ] += d_areae[ie] / 4.0 ;
  d_areav[ iv[1] ] += d_areae[ie] / 4.0 ;
  d_areav[ iv[2] ] += d_areae[ie] / 4.0 ;
  d_areav[ iv[3] ] += d_areae[ie] / 4.0 ;
});
```

The main difference for the HIP variant is that we use explicit device memory allocation/deallocation and host-device memory copy operations.

Sum Reduction: Vector Dot Product

This section contains an exercise file `RAJA/exercises/dot-product.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/dot-product_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make dot-product` and `make dot-product_solution` from the build directory.

Key RAJA features shown in this example are:

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

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

```
double dot = 0.0;

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

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

Although this kernel is serial, it is representative of a *reduction* operation which is a common algorithm 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 direct support for reductions 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 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 allows multiple reduction operations to be performed in a single kernel alongside other computations. Please see *Reduction Operations* for more information.

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

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

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

dot = seqdot.get();
```

The sum reduction object is defined by specifying the reduction policy `RAJA::seq_reduce` matching the kernel execution policy `RAJA::seq_exec`, 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 kernel 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 kernel execution policy.

The RAJA CUDA variant is achieved by using appropriate kernel 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 kernel 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) {
  hpdot += d_a[i] * d_b[i];
});

dot = hpdot.get();
```

It is worth repeating 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.

Note: Currently available reduction capabilities in RAJA require a *reduction policy* type that is compatible with the execution policy for the kernel in which the reduction is used. We are developing a new reduction interface for RAJA that will provide an alternative for which the reduction policy is not required.

Reduction Types and Kernels with Multiple Reductions

This section contains an exercise file `RAJA/exercises/reductions.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/reductions_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make reductions` and `make reductions_solution` from the build directory.

Key RAJA features shown in this section are:

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

In the *Sum Reduction: Vector Dot Product* exercise, 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: RAJA 'min-loc' and 'max-loc' reductions determine the min and max reduction value, respectively, along with an iteration index at which the main/max value is found.

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.

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.

We start by allocating an array 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
//
  constexpr int N = 1000000;

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

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

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

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

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

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

With these parameters and data initialization, the code example 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 along with operations after the kernel to print the reduced values 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);

  });

  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 for the type of reduction (e.g., '+=' 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 exercise can be run with the execution and reduction policies:

```
  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;
```

Atomic Operations: Computing a Histogram

This section contains an exercise file `RAJA/exercises/atomic-histogram.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/atomic-histogram_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make atomic-histogram` and `make atomic-histogram_solution` from the build directory.

Key RAJA features shown in this exercise are:

- `RAJA::forall` kernel execution template and execution policies
- `RAJA::TypedRangeSegment` iteration space construct
- RAJA atomic add operation and RAJA atomic operation policies

The example uses an integer array of length 'N' randomly initialized with values in the interval [0, M).

```
constexpr int M = 20;
constexpr int N = 100000;

int* array = memoryManager::allocate<int>(N);
int* hist = memoryManager::allocate<int>(M);

for (int i = 0; i < N; ++i) {
  array[i] = rand() % M;
}
```

Each kernel iterates over the array and accumulates the number of occurrences of each value in [0, M) in another array named 'hist'. The kernels use atomic operations for the accumulation, which 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 atomic operation requires an atomic policy type parameter that must be compatible with the execution policy for the kernel in which it is used. This is similar to the reduction policies we described in *Sum Reduction: Vector Dot Product*.

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

All code snippets described below use the stride-1 iteration space range:

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

Here is the OpenMP version:

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

(continues on next page)

(continued from previous page)

```
    RAJA::atomicAdd<RAJA::omp_atomic>(&hist[array[i]], 1);

  });
```

One is added to a slot in the 'bins' array 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 kernel 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>(&hist[array[i]], 1);

  });
```

and:

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

    RAJA::atomicAdd<RAJA::hip_atomic>(&hist[array[i]], 1);

  });
```

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

Note that RAJA provides an `auto_atomic` policy for easier usage and improved portability. This policy will choose the proper atomic operation for the execution policy used to run the kernel. Specifically, when OpenMP is enabled, the OpenMP atomic policy will be used, which is correct in a sequential or OpenMP execution context. 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>(&hist[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>(&hist[array[i]], 1);

  });
```

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

Parallel Scan Operations

This section contains an exercise file `RAJA/exercises/scan.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/scan_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make scan` and `make scan_solution` from the build directory.

Key RAJA features shown in this section are:

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

In this section, we present examples of various RAJA scan operations using multiple RAJA execution back-ends. 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 *Scan Operations*.

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

Note: RAJA scan operations take 'span' arguments to express the sequential index range of array entries used in the scan. Typically, these span objects are created using the `RAJA::make_span` method as shown in the examples below.

Each of the examples below uses the same integer arrays for input and output values. We initialize the input array and print its values as such:

```
//
// Define array length
//
  constexpr 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);

  std::cout << "\n in values...\n";
  printArray(in, N);
  std::cout << "\n";
```

This generates the following sequence of values. This sequence will be used as the 'in' array for each of the following examples.:

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

Inclusive Scans

RAJA's scan operations are standalone operations. That is, they cannot be combined with other operations in a kernel. A sequential inclusive scan operation can be executed like so:

```
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 'plus' 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:

```
-1 -1 0 2 5 9 14 20 27 35 44 54 65 77 90 104 119 135 152 170
```

In particular, each entry in the output array is a *partial sum* of all input array entries up to that array index.

We can be explicit about the operation used in the scan by passing the RAJA 'plus' operator `RAJA::operators::plus<int>` 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 as above.

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 expected, this produces the same result as the previous two examples.

As is commonly the case with RAJA, the only difference between this code and the previous one is the execution policy. If we want to run the scan on a GPU using CUDA, we would use a CUDA execution policy as is shown in examples below.

Note: If no operator is passed to a RAJA scan operation, the default plus operator is used, resulting in a prefix-sum.

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 -1 -1 0 2 5 9 14 20 27 35 44 54 65 77 90 104 119 135 152
```

The result of an exclusive scan is similar to the result of an inclusive scan, but differs in two ways. First, the first entry in the exclusive scan output array is the *identity* of the operator used. In the example here, it is zero, since the operator is 'plus'. Second, the output sequence is shifted one position to the right when compared to an inclusive scan.

Note: The *identity* of an operator is the default value of a given type for that operation. For example: - The identity of an int for a sum operation is 0. - The identity of an int for a maximum operation is -2147483648.

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 when using `RAJA::forall`.

In-place Scans and Other Operators

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

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

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

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

Note that, before the scan operation is invoked, we copy the input array into the output array to provide the scan input array we want.

This generates the following sequence in the output array:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Since the operator used in the scan is 'minimum' and the smallest values in the input array is the first entry, the result is an array with that value in all array slots.

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

Since it is an exclusive scan, the first value in the result is the negative of the max int value, which is 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 -1 -1 0 2 5 9 14 20 27 35 44 54 65 77 90 104 119 135 152
```

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>{});
```

and the same using the RAJA HIP back-end:

```
  RAJA::inclusive_scan_inplace<RAJA::hip_exec<HIP_BLOCK_SIZE>>(
      RAJA::make_span(d_out, N),
      RAJA::operators::plus<int>{});
```

Parallel Sort Operations

This section contains an exercise file `RAJA/exercises/sort.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/sort_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make sort` and `make sort_solution` from the build directory.

Key RAJA features shown in this section are:

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

We show examples of RAJA sort operations using multiple RAJA execution back-ends and describe 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 available RAJA sorts, please see *Sort Operations*.

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

Note: RAJA sort operations take 'span' arguments to express the sequential index range of array entries used in the sort. Typically, these span objects are created using the `RAJA::make_span` method as shown in the examples below.

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
//
  constexpr int N = 20;

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

(continues on next page)

(continued from previous page)

```
  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);

  std::cout << "\n in keys...\n";
  printArray(in, N);
  std::cout << "\n in (key, value) pairs...\n";
  printArray(in, in_vals, N);
  std::cout << "\n";
```

This produces 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 shown as pairs of values in the `in` and `in_vals` arrays, respectively:

```
(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)
```

Note: In the following sections, we discuss *stable* and *unstable* sort operations. The difference between them is that a stable sort preserves the relative order of equal elements, with respect to the sort comparator operation, while an unstable sort may not preserve the relative order of equal elements. For the examples below that use integer arrays, there is no way to tell by inspecting the output whether relative ordering is preserved for unstable sorts. However, the preservation of relative ordering can be seen in the sort pairs examples below.

Unstable Sorts

A sequential unstable sort operation is performed by:

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

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

Since no comparator is passed to the sort method, the default 'less' operator `RAJA::operators::less<int>` is applied and the result generated in the `out` array is a non-decreasing sequence of values from the `in` array; i.e.,:

```
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 operation by passing the 'less' operator to the sort method manually:

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

The result in the `out` array is the same as before:

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

An unstable parallel sort operation using OpenMP multithreading is accomplished similarly by replacing the execution policy type with and OpenMP policy:

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

As is common 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 or HIP, we would use a CUDA or HIP execution policy. This is shown in examples that follow.

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 as expected based on the examples we discussed above:

```
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 an array of 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 when using `RAJA::forall`.

Other Comparators

Using a different comparator operator allows sorting in a different order. Here is a sequential stable sort that uses the 'greater' operator `RAJA::operators::greater<int>`:

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

and similarly for HIP:

```
RAJA::stable_sort<RAJA::hip_exec<HIP_BLOCK_SIZE>>(
  RAJA::make_span(d_out, N),
  RAJA::operators::greater<int>{});
```

Both of these sorts generate 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:

- The only operators provided by RAJA that are valid to use in sort because they enforce a strict weak ordering of elements for arithmetic types are 'less' and 'greater'. Users may provide other operators for different sorting operations.
- Also the RAJA CUDA sort back-end only supports RAJA operators 'less' and 'greater' because it uses the NVIDIA CUB library.

Sort Pairs

Sort pairs operations generate the same results as the sort operations we have just described. Additionally, a second array of values is reordered using the ordering of the first sorted array so **two arrays are passed to sort pairs methods.**

Note: For `RAJA::sort_pairs` algorithms, two arrays are passed. The first array (*keys*) will be sorted according to the given comparator operator. The elements in the second array (*values*) will be reordered based on the final order of the first sorted array.

Here is a sequential unstable sort pairs operation 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 that they appeared in the unsorted arrays like `(8,0) (8,1)`, while others are reversed like `(9,1) (9,0)`. This illustrates that relative ordering of equal elements may not be preserved in an unstable sort.

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 input 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 only difference is that a HIP execution policy template parameter type is used.

Data Views and Layouts

This section contains an exercise file `RAJA/exercises/view-layout.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/view-layout_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make view-layout` and `make view-layout_solution` from the build directory.

Key RAJA features shown in this section are:

- `RAJA::View`
- `RAJA::Layout` and `RAJA::OffsetLayout` constructs
- Layout permutations

The examples in this section illustrate RAJA View and Layout concepts and usage patterns. The goal is for you to gain an understanding of how to use RAJA Views and Layouts to simplify and transform array data access patterns. None of the examples use RAJA kernel execution methods, such as `RAJA::forall`. The intent is to focus on RAJA View and Layout mechanics.

Consider a basic C-style implementation of a matrix-matrix multiplication operation, using $N \times N$ matrices:

```
for (int row = 0; row < N; ++row) {
  for (int col = 0; col < N; ++col) {
    for (int k = 0; k < N; ++k) {
      Cref[col + N*row] += A[k + N*row] * B[col + N*k];
    }
  }
}
```

As is commonly done for efficiency in C and C++, we have allocated the data for the matrices as one-dimensional arrays. Thus, we need to manually compute the data pointer offsets for the row and column indices in the kernel. Here, we use the array `Cref` to hold a reference solution matrix that we use to compare with results generated by the examples below.

To simplify the multi-dimensional indexing, we can use `RAJA::View` objects, which we define as:

```
RAJA::View< double, RAJA::Layout<2, int> > Aview(A, N, N);
RAJA::View< double, RAJA::Layout<2, int> > Bview(B, N, N);
RAJA::View< double, RAJA::Layout<2, int> > Cview(C, N, N);
```

Here we define three `RAJA::View` objects, 'Aview', 'Bview', and 'Cview', that *wrap* the array data pointers, 'A', 'B', and 'C', respectively. We pass a data pointer as the first argument to each view constructor and then the extent of each matrix dimension as the second and third arguments. There are two extent arguments since we indicate in the `RAJA::Layout` template parameter list. The matrices are square and each extent is 'N'. Here, the template parameters to `RAJA::View` are the array data type 'double' and a `RAJA::Layout` type. Specifically:

```
RAJA::Layout<2, int>
```

means that each View represents a two-dimensional default data layout, and that we will use values of type 'int' to index into the arrays.

Using the `RAJA::View` objects, we can access the data entries for the rows and columns using a more natural, less error-prone syntax:

```
  for (int row = 0; row < N; ++row) {
    for (int col = 0; col < N; ++col) {
      for (int k = 0; k < N; ++k) {
        Cview(row, col) += Aview(row, k) * Bview(k, col);
      }
    }
  }
```

Default Layouts Use Row-major Ordering

The default data layout ordering in RAJA is *row-major*, which is the convention for multi-dimensional array indexing in C and C++. This means that the rightmost index will be stride-1, the index to the left of the rightmost index will have stride equal to the extent of the rightmost dimension, and so on.

Note: RAJA Layouts and Views support any number of dimensions and the default data access ordering is *row-major*. Please see *View and Layout* for more details.

To illustrate the default data layout striding, we next show simple one-, two-, and three-dimensional examples where the for-loop ordering for the different dimensions is such that all data access is stride-1. We begin by defining some dimensions, allocate and initialize arrays:

```
  constexpr int Nx = 3;
  constexpr int Ny = 5;
  constexpr int Nz = 2;
  constexpr int Ntot  = Nx*Ny*Nz;
  int* a = new int[ Ntot ];
  int* aref = new int[ Ntot ];

  for (int i = 0; i < Ntot; ++i)
  {
    aref[i] = i;
  }
```

The version of the array initialization kernel using a one-dimensional `RAJA::View` is:

```
  RAJA::View< int, RAJA::Layout<1, int> > view_1D(a, Ntot);

  for (int i = 0; i < Ntot; ++i) {
    view_1D(i) = i;
  }
```

The version of the array initialization using a two-dimensional `RAJA::View` is:

```
RAJA::View< int, RAJA::Layout<2, int> > view_2D(a, Nx, Ny);

int iter{0};
for (int i = 0; i < Nx; ++i) {
  for (int j = 0; j < Ny; ++j) {
    view_2D(i, j) = iter;
    ++iter;
  }
}
```

The three-dimensional version is:

```
RAJA::View< int, RAJA::Layout<3, int> > view_3D(a, Nx, Ny, Nz);

iter = 0;
for (int i = 0; i < Nx; ++i) {
  for (int j = 0; j < Ny; ++j) {
    for (int k = 0; k < Nz; ++k) {
      view_3D(i, j, k) = iter;
      ++iter;
    }
  }
}
```

It's worth repeating that the data array access in all three variants shown here using `RAJA::View` objects is stride-1 since we order the for-loops in the loop nests to match the row-major ordering.

RAJA Layout types support other data access patterns with different striding orders, offsets, and permutations. To this point, we have used the default Layout constructor. RAJA provides methods to generate Layouts for different indexing patterns. We describe these in the next several sections. Next, we show how to permute the data striding order using permuted Layouts.

Permuted Layouts Change Data Striding Order

Every `RAJA::Layout` object has a permutation. When a permutation is not specified at creation, a Layout will use the identity permutation. Here are examples where the identity permutation is explicitly provided. First, in two dimensions:

```
std::array<RAJA::idx_t, 2> defperm2 {{0, 1}};
RAJA::Layout< 2, int > defperm2_layout =
  RAJA::make_permuted_layout( {{Nx, Ny}}, defperm2);
RAJA::View< int, RAJA::Layout<2, int> > defperm_view_2D(a, defperm2_layout);

iter = 0;
for (int i = 0; i < Nx; ++i) {
  for (int j = 0; j < Ny; ++j) {
    defperm_view_2D(i, j) = iter;
    ++iter;
  }
}
```

Then, in three dimensions:

```
std::array<RAJA::idx_t, 3> defperm3 {{0, 1, 2}};
RAJA::Layout< 3, int > defperm3_layout =
  RAJA::make_permuted_layout( {{Nx, Ny, Nz}}, defperm3);
RAJA::View< int, RAJA::Layout<3, int> > defperm_view_3D(a, defperm3_layout);

iter = 0;
for (int i = 0; i < Nx; ++i) {
  for (int j = 0; j < Ny; ++j) {
    for (int k = 0; k < Nz; ++k) {
      defperm_view_3D(i, j, k) = iter;
      ++iter;
    }
  }
}
```

These two examples access the data with stride-1 ordering, the same as in the earlier examples, which is shown by the nested loop ordering. The identity permutation in two dimensions is '{0, 1}' and is '{0, 1, 2}' for three dimensions. The method `RAJA::make_permuted_layout` is used to create a `RAJA::Layout` object with a permutation. The method takes two arguments, the extents of each dimension and the permutation.

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

---

Next, we permute the striding order for the two-dimensional example:

```
std::array<RAJA::idx_t, 2> perm2 {{1, 0}};
RAJA::Layout< 2, int > perm2_layout =
  RAJA::make_permuted_layout( {{Nx, Ny}}, perm2);
RAJA::View< int, RAJA::Layout<2, int> > perm_view_2D(a, perm2_layout);

iter = 0;
for (int j = 0; j < Ny; ++j) {
  for (int i = 0; i < Nx; ++i) {
    perm_view_2D(i, j) = iter;
    ++iter;
  }
}
```

Read from right to left, the permutation '{1, 0}' specifies that the first (zero) index 'i' is stride-1 and the second index (one) 'j' has stride equal to the extent of the first Layout dimension 'Nx'. This is evident in the for-loop ordering.

Here is the three-dimensional case, where we have reversed the striding order using the permutation '{2, 1, 0}':

```
std::array<RAJA::idx_t, 3> perm3a {{2, 1, 0}};
RAJA::Layout< 3, int > perm3a_layout =
  RAJA::make_permuted_layout( {{Nx, Ny, Nz}}, perm3a);
RAJA::View< int, RAJA::Layout<3, int> > perm3a_view_3D(a, perm3a_layout);

iter = 0;
for (int k = 0; k < Nz; ++k) {
  for (int j = 0; j < Ny; ++j) {
    for (int i = 0; i < Nx; ++i) {
      perm3a_view_3D(i, j, k) = iter;
      ++iter;
    }
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

The data access remains stride-1 due to the for-loop reordering. For fun, here is another three-dimensional permutation:

```
  std::array<RAJA::idx_t, 3> perm3b {{1, 2, 0}};
  RAJA::Layout< 3, int > perm3b_layout =
    RAJA::make_permuted_layout( {{Nx, Ny, Nz}}, perm3b);
  RAJA::View< int, RAJA::Layout<3, int> > perm3b_view_3D(a, perm3b_layout);

  iter = 0;
  for (int j = 0; j < Ny; ++j) {
    for (int k = 0; k < Nz; ++k) {
      for (int i = 0; i < Nx; ++i) {
        perm3b_view_3D(i, j, k) = iter;
        ++iter;
      }
    }
  }
```

The permutation is '{1, 2, 0}' so to make the data access stride-1, we swap the 'j' and 'k' loops and leave the 'i' loop as the inner loop.

Multi-dimensional Indices and Linear Indices

`RAJA::Layout` types provide methods to convert between linear indices and multi-dimensional indices and vice versa. Recall the Layout 'perm3a_layout' from above that was created with the permutation '{2, 1, 0}'. To get the linear index corresponding to the index triple '(1, 2, 0)', you can do this:

```
int lin = perm3a_layout(1, 2, 0);
```

The value of 'lin' is 7 = 1 + 2 * Nx + 0 * Nx * Ny. To get the index triple for linear index 7, you can do:

```
int i, j, k;
perm3a_layout.toIndices(7, i, j, k);
```

This sets 'i' to 1, 'j' to 2, and 'k' to 0.

Similarly for the Layout 'permb_layout', which was created with the permutation '{1, 2, 0}':

```
lin = perm3b_layout(1, 2, 0);
```

sets 'lin' to 13 = 1 + 0 * Nx + 2 * Nx * Nz and:

```
perm3b_layout.toIndices(13, i, j, k);
```

sets 'i' to 1, 'j' to 2, and 'k' to 0.

There are more examples in the exercise file associated with this section. Feel free to experiment with them.

One important item to note is that, by default, there is no bounds checking on indices passed to a `RAJA::View` data access method or `RAJA::Layout` index computation methods. Therefore, it is the responsibility of a user to ensure that indices passed to `RAJA::View` and `RAJA::Layoout` methods are in bounds to avoid accessing data outside of the View or computing invalid indices.

Note: RAJA provides a CMake variable `RAJA_ENABLE_BOUNDS_CHECK` to turn run time bounds checking on or off when the code is compiled. Enabling bounds checking is useful for debugging and to ensure your code is correct. However, when enabled, bounds checking adds noticeable run time overhead. So it should not be enabled for a production build of your code.

Offset Layouts Apply Offsets to Indices

The last topic we cover in this exercise is the `RAJA::OffsetLayout` type. We first illustrate the concept of an offset with a C-style for-loop:

```
int imin = -5;
int imax = 6;

for (int i = imin; i < imax; ++i) {
  ao_ref[ i-imin ] = i;
}
```

Here, the for-loop runs from 'imin' to 'imax-1' (i.e., -5 to 5). To avoid out-of-bounds negative indexing, we subtract 'imin' (i.e., -5) from the loop index 'i'.

To do the same thing with RAJA, we create a `RAJA::OffsetLayout` object and use it to index into the array:

```
RAJA::OffsetLayout<1, int> offlayout_1D =
  RAJA::make_offset_layout<1, int>( {{imin}}, {{imax}} );

RAJA::View< int, RAJA::OffsetLayout<1, int> > aoview_1Doff(ao,
                                                           offlayout_1D);

for (int i = imin; i < imax; ++i) {
  aoview_1Doff(i) = i;
}
```

`RAJA::OffsetLayout` is a different type than `RAJA::Layout` because it contains offset information. The arguments to the `RAJA::make_offset_layout` method are the index bounds.

As expected, the two dimensional case is similar. First, a C-style loop:

```
imin = -1;
imax = 2;
int jmin = -5;
int jmax = 5;

iter = 0;
for (int i = imin; i < imax; ++i) {
  for (int j = jmin; j < jmax; ++j) {
    ao_ref[ (j-jmin) + (i-imin) * (jmax-jmin)  ] = iter;
    iter++;
  }
}
```

and then the same operation using a `RAJA::OffsetLayout` object:

```
RAJA::OffsetLayout<2, int> offlayout_2D =
  RAJA::make_offset_layout<2, int>( {{imin, jmin}}, {{imax, jmax}} );
```

(continues on next page)

(continued from previous page)

```
RAJA::View< int, RAJA::OffsetLayout<2, int> > aoview_2Doff(ao,
                                                           offlayout_2D);
iter = 0;
for (int i = imin; i < imax; ++i) {
  for (int j = jmin; j < jmax; ++j) {
    aoview_2Doff(i, j) = iter;
    iter++;
  }
}
```

Note that the first argument passed to `RAJA::make_offset_layout` contains the lower bounds for 'i' and 'j' and the second argument contains the upper bounds. Also, the 'j' index is stride-1 by default since we did not pass a permutation to the `RAJA::make_offset_layout` method, which is the same as the non-offset Layout usage.

Just like `RAJA::Layout` has a permutation, so does `RAJA::OffsetLayout`. Here is an example where we permute the (i, j) index stride ordering:

```
std::array<RAJA::idx_t, 2> perm1D {{1, 0}};
RAJA::OffsetLayout<2> permofflayout_2D =
  RAJA::make_permuted_offset_layout<2>( {{imin, jmin}},
                                        {{imax, jmax}},
                                        perm1D );

RAJA::View< int, RAJA::OffsetLayout<2> > aoview_2Dpermoff(ao,
                                                          permofflayout_2D);

iter = 0;
for (int j = jmin; j < jmax; ++j) {
  for (int i = imin; i < imax; ++i) {
    aoview_2Dpermoff(i, j) = iter;
    iter++;
  }
}
```

The permutation '{1, 0}' is passed as the third argument to `RAJA::make_offset_layout`. From the ordering of the for-loops, we can see that the 'i' index is stride-1 and the 'j' index has stride equal to the extent of the 'i' dimension so the for-loop nest strides through the data with unit stride.

Permuted Layout: Batched Matrix-Multiplication

This section contains an exercise file `RAJA/exercises/permuted-layout-batch-matrix-multiply.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/permuted-layout-batch-matrix-multiply_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make permuted-layout-batch-matrix-multiply` and `make permuted-layout-batch-matrix-multiply_solution` from the build directory.

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 a "batched" matrix multiplication operation for a collection of 3×3 matrices. Each pair of matrices A^e and B^e, indexed by 'e', is multiplied and the product is stored in a matrix C^e. A^e matrix entries, for all values of e, are stored in an array A, all B^e matrices are stored in an array B, and all C^e matrices are stored in an array C. In the following discussion, the notation A^e_{rc} indicates the row r and column c entry of the 3×3 matrix A^e.

In the exercise, we use two different data layouts for the arrays A, B, and C to represent different storage patterns for the 3×3 matrices. Below, we describe these layouts for two 3×3 matrices. The extension to more than two matrices is straightforward as you will see in the exercise code. In the exercise code, we time the execution of the batched matrix multiplication operation to compare the performance for each layout and execution policy. These comparisons are not completely conclusive as to which layout is best since there may be additional performance to be gained by more specific tuning of the memory layouts for an architecture and execution back-end. A complete, detailed analysis of the performance implications of memory layout and access patterns is beyond the scope of the exercise.

In **layout 1**, the entries for each 3×3 matrix are contiguous in memory following row major ordering; i.e., the ordering is column index, then row index, then matrix index:

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

In **layout 2**, the matrix entries are first ordered by matrix index, then by column index, and finally by row index:

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

Permuted Layouts

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

The views to access data 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, int, 2>> Aview(A, layout1);
RAJA::View<double, RAJA::Layout<3, int, 2>> Bview(B, layout1);
RAJA::View<double, RAJA::Layout<3, int, 2>> Cview(C, layout1);
```

The first argument to `RAJA::make_permuted_layout` is an array whose entries correspond to the extent of each layout dimension. Here, we have N $N_r \times N_c$ matrices. The second argument, the layout permutation, describes the striding order of the array indices. 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 N_c, the number of columns in each matrix. The matrix index (index 0) has stride $N_r \times N_c$, the number of entries in each matrix.

The views for layout 2 are constructed similarly, with a different index striding order:

```
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, int, 0>> Aview2(A2, layout2);
RAJA::View<double, RAJA::Layout<3, int, 0>> Bview2(B2, layout2);
RAJA::View<double, RAJA::Layout<3, int, 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 with the same 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) have stride N, which is the number of matrices, and the row index (index 1) has stride $N \times N_c$.

RAJA Kernel Variants

The exercise files contain RAJA variants that run the batched matrix multiplication kernel with different execution back-ends. As mentioned earlier, we print out execution timings for each so you can compare the run times of the different layouts described above. For example, the sequential CPU variant using layout 1 is:

```
  RAJA::forall<RAJA::loop_exec>(RAJA::TypedRangeSegment<int>(0, N),
    [=](int 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 sequential CPU variant using layout 2 is:

```
  RAJA::forall<RAJA::loop_exec>(RAJA::TypedRangeSegment<int>(0, N),
    [=](int e) {

      Cview2(e, 0, 0) = Aview2(e, 0, 0) * Bview2(e, 0, 0)
                      + Aview2(e, 0, 1) * Bview2(e, 1, 0)
                      + Aview2(e, 0, 2) * Bview2(e, 2, 0);
      Cview2(e, 0, 1) = Aview2(e, 0, 0) * Bview2(e, 0, 1)
                      + Aview2(e, 0, 1) * Bview2(e, 1, 1)
```

(continues on next page)

(continued from previous page)

```
                        + Aview2(e, 0, 2) * Bview2(e, 2, 1);
      Cview2(e, 0, 2) = Aview2(e, 0, 0) * Bview2(e, 0, 2)
                        + Aview2(e, 0, 1) * Bview2(e, 1, 2)
                        + Aview2(e, 0, 2) * Bview2(e, 2, 2);

      Cview2(e, 1, 0) = Aview2(e, 1, 0) * Bview2(e, 0, 0)
                        + Aview2(e, 1, 1) * Bview2(e, 1, 0)
                        + Aview2(e, 1, 2) * Bview2(e, 2, 0);
      Cview2(e, 1, 1) = Aview2(e, 1, 0) * Bview2(e, 0, 1)
                        + Aview2(e, 1, 1) * Bview2(e, 1, 1)
                        + Aview2(e, 1, 2) * Bview2(e, 2, 1);
      Cview2(e, 1, 2) = Aview2(e, 1, 0) * Bview2(e, 0, 2)
                        + Aview2(e, 1, 1) * Bview2(e, 1, 2)
                        + Aview2(e, 1, 2) * Bview2(e, 2, 2);

      Cview2(e, 2, 0) = Aview2(e, 2, 0) * Bview2(e, 0, 0)
                        + Aview2(e, 2, 1) * Bview2(e, 1, 0)
                        + Aview2(e, 2, 2) * Bview2(e, 2, 0);
      Cview2(e, 2, 1) = Aview2(e, 2, 0) * Bview2(e, 0, 1)
                        + Aview2(e, 2, 1) * Bview2(e, 1, 1)
                        + Aview2(e, 2, 2) * Bview2(e, 2, 1);
      Cview2(e, 2, 2) = Aview2(e, 2, 0) * Bview2(e, 0, 2)
                        + Aview2(e, 2, 1) * Bview2(e, 1, 2)
                        + Aview2(e, 2, 2) * Bview2(e, 2, 2);

    }
  );
```

The only differences between these two are the names of the views that appear in the lambda expression loop body since a different layout is used to create view objects for each layout case. To make the algorithm code identical for all cases, we could use type aliases for the view and layout types in a header file similar to how we may abstract the execution policy out of the algorithm, and compile the code for the case we want to run.

For comparison, here is an OpenMP CPU variant using layout 1:

```
  RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::TypedRangeSegment<int>(0, N),
    [=](int 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);
```

(continues on next page)

(continued from previous page)

```
      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 difference between this variant and the sequential CPU variant shown above is the execution policy. The lambda expression loop body is identical to the sequential CPU variant.

The exercise files also contain variants for RAJA CUDA and HIP back-ends. Their similarities and differences are the same as what we've just described.

Complex Loops and Advanced RAJA Features

RAJA provides two APIs for writing complex kernels involving nested loops: `RAJA::kernel` that has been available for several years and `RAJA::expt::launch`, which is more recent and which will be moved out of the `expt` namespace soon. We briefly introduce both interfaces here. The tutorial sections that follow provide much more detailed descriptions.

`RAJA::kernel` is analogous to `RAJA::forall` in that it involves kernel execution templates, execution policies, iteration spaces, and lambda expression kernel bodies. The main differences between `RAJA::kernel` and `RAJA::forall` are:

- `RAJA::kernel` requires a tuple of iteration spaces, one for each level in a loop nest, whereas `RAJA::forall` takes exactly one iteration space.
- `RAJA::kernel` can accept multiple lambda expressions to express different parts of a kernel body, whereas `RAJA::forall` accepts exactly one lambda expression for a kernel body.
- `RAJA::kernel` execution policies are more complicated than those for `RAJA::forall`. `RAJA::forall` policies essentially represent the kernel execution back-end only. `RAJA::kernel` execution policies enable complex compile time algorithm transformations to be done without changing the kernel code.

The following exercises illustrate the common usage of `RAJA::kernel` and ``RAJA::expt::launch`. Please see *RAJA Kernel Execution Policies* for more information about other execution policy constructs `RAJA::kernel` provides. `RAJA::expt::launch` takes a `RAJA::expt::Grid` type argument for representing a teams-thread launch configuration, and a lambda expression which takes a `RAJA::expt::LaunchContext` argument. `RAJA::expt::launch` allows an optional run time choice of execution environment, either CPU or GPU. Code written inside the lambda expression body will execute in the chosen execution environment. Within that environment, a user executes kernel operations using `RAJA::expt::loop<EXEC_POL>` method calls, which take lambda expressions to express loop body operations.

Note: A key difference between the `RAJA::kernel` and `RAJA::expt::launch` approaches is that almost all of the kernel execution pattern is expressed in the execution policy when using `RAJA::kernel`, whereas with `RAJA::expt::launch` the kernel execution pattern is expressed mostly in the lambda expression kernel body.

One may argue that RAJA::kernel is more portable and flexible in that the execution policy enables compile time code transformations without changing kernel body code. On the other hand, RAJA::expt::launch is less opaque and more intuitive, but may require kernel body code changes for algorithm changes. Which interface to use depends on personal preference and other concerns, such as portability requirements, the need for run time execution selection, etc. Kernel structure is more explicit in application source code with RAJA::expt::launch, and more concise and arguably more opaque with RAJA::kernel. There is a large overlap of algorithms that can be expressed with either interface. However, there are things that one can do with one or the other but not both.

In the following sections, we introduce the basic mechanics and features of both APIs with examples and exercises. We also present a sequence of execution policy examples and matrix transpose examples using both RAJA::kernel and RAJA::expt::launch to compare and contrast the two interfaces.

Nested Loops with RAJA::kernel

The examples in this section illustrate various features of the RAJA::kernel API used to execute nested loop kernels. It describes how to construct kernel execution policies and use different view types and tiling mechanisms to transform loop patterns. More information can be found in *Complex Loops (RAJA::kernel)*.

Basic RAJA::kernel Mechanics and Nested Loop Ordering

This section contains an exercise file RAJA/exercises/kernelintro-nested-loop-reorder.cpp for you to work through if you wish to get some practice with RAJA. The file RAJA/exercises/kernelintro-nested-loop-reorder_solution.cpp contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute make kernelintro-nested-loop-reorder and make kernelintro-nested-loop-reorder_solution from the build directory.

Key RAJA features shown in this section are:

- RAJA::kernel loop iteration templates and execution policies
- Nested loop reordering
- RAJA strongly-types indices

The examples in this section show the nested loop reordering process in more detail. Specifically, we describe how to reorder execution policy statements, which is conceptually analogous to how one would reorder for-loops in a C-style loop nest. We also introduce strongly-typed index variables that can help users write correct nested loop code with RAJA. The examples do not perform any 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 to avoid complications resulting from print statements used in parallel programs. The mechanics shown here work the same way for parallel RAJA execution policies.

Before we dive into code, we reiterate important features that represent the main differences between nested-loop RAJA and the RAJA::forall construct for simple, non-nested loop kernels:

- 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 below.
- 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 (i, j, k):

```
RAJA_INDEX_VALUE_T(KIDX, int, "KIDX");
RAJA_INDEX_VALUE_T(JIDX, int, "JIDX");
RAJA_INDEX_VALUE_T(IIDX, int, "IIDX");
```

Specifically, the 'i' index variable type is `IIDX`, the 'j' index variable is `JIDX`, and the 'k' variable is `KIDX`, which are aliases to `int` type.

We also define [min, max) intervals for each loop index:

```
  constexpr int imin = 0;
  constexpr int imax = 2;
  constexpr int jmin = 1;
  constexpr int jmax = 3;
  constexpr int kmin = 2;
  constexpr int kmax = 4;
```

and three corresponding **typed** range segments which bind the ranges to the index variable types via template specialization:

```
  RAJA::TypedRangeSegment<KIDX> KRange(kmin, kmax);
  RAJA::TypedRangeSegment<JIDX> JRange(jmin, jmax);
  RAJA::TypedRangeSegment<IIDX> IRange(imin, imax);
```

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. This is illustrated at the end of this section.

We begin with a C-style loop nest with 'i' in the inner loop, 'j' in the middle loop, and 'k' in the outer loop, which prints the (i, j, k) triple in the inner loop body:

```
  for (int k = kmin; k < kmax; ++k) {
    for (int j = jmin; j < jmax; ++j) {
      for (int i = imin; i < imax; ++i) {
        printf( " (%d, %d, %d) \n", i, j, k);
      }
    }
  }
```

The `RAJA::kernel` version of this is:

```
  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) {
```

(continues on next page)

(continued from previous page)

```
    printf( " (%d, %d, %d) \n", (int)(*i), (int)(*j), (int)(*k));
 });
```

The integer template parameters in the `RAJA::statement::For` types represent the lambda expression index argument and the range types in the iteration space tuple argument to `RAJA::kernel`.

Both kernels generate the same output, as expected:

```
(I, J, K)
---------
(0, 1, 2)
(1, 1, 2)
(0, 2, 2)
(1, 2, 2)
(0, 1, 3)
(1, 1, 3)
(0, 2, 3)
(1, 2, 3)
```

which you can see by running the exercise code.

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. This is required for the code to compile.

Note: RAJA provides mechanisms to explicitly specify which loop variables, for example, and in which order they appear in a lambda expression argument list. Please refer to *Complex Loops (RAJA::kernel)* for more information.

The execution policy for the loop nest is specified in the `RAJA::KernelPolicy` type. The policy uses two statement types: `RAJA::statement::For` and `RAJA::statement::Lambda`.

The `RAJA::statement::Lambda` is used to generate code that invokes the lambda expression. The '0' template parameter refers to the index of the lambda expression in the `RAJA::kernel` argument list following the iteration space tuple. Since there is only one lambda expression, we reference it with the '0' identifier. Sometimes more complicated kernels require multiple lambda expressions, so we need a way to specify where they will appear in the generated executable code. We show examples of this in the matrix transpose discussion later in the tutorial.

Each level in the loop nest is identified by a `RAJA::statement::For` type, which identifies the iteration space and execution policy for the level. Here, each level uses a sequential execution policy, which is for illustration purposes. The integer that appears as the first template argument to each `RAJA::statement::For` type corresponds to the index of a range in the tuple and also to the associated lambda index argument; i.e., '0' for 'i', '1' for 'j', and '2' for 'k'.

The integer argument to each `RAJA::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. To illustrate, 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>
                          >
                        >
```

(continues on next page)

(continued from previous page)

```
                        >
                      >;

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

This generates the following output:

```
(I, J, K)
---------
(0, 1, 2)
(0, 1, 3)
(1, 1, 2)
(1, 1, 3)
(0, 2, 2)
(0, 2, 3)
(1, 2, 2)
(1, 2, 3)
```

which is the same as the corresponding C-style version:

```
  for (int j = jmin; j < jmax; ++j) {
    for (int i = imin; i < imax; ++i) {
      for (int k = kmin; k < kmax; ++k) {
        printf( " (%d, %d, %d) \n", i, j, k);
      }
    }
  }
```

Note that we have simply reordered the nesting of the `RAJA::statement::For` types in the execution policy. This is analogous to reordering the for-loops in C-style version.

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>
                            >
                          >
                        >
                      >;

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

The analogous C-style loop nest is:

```
  for (int i = imin; i < imax; ++i) {
    for (int k = kmin; k < kmax; ++k) {
      for (int j = jmin; j < jmax; ++j) {
```

(continues on next page)

(continued from previous page)

```
        printf( " (%d, %d, %d) \n", i, j, k);
      }
    }
  }
```

The output generated by these two kernels is:

```
(I, J, K)
---------
(0, 1, 2)
(0, 2, 2)
(0, 1, 3)
(0, 2, 3)
(1, 1, 2)
(1, 2, 2)
(1, 1, 3)
(1, 2, 3)
```

Finally, we show an example that will generate a compilation error because there is a type mismatch in the ordering of the range segments in the tuple and the lambda expression argument list.

```
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));
});
```

Do you see the problem? The last kernel is included in the exercise source file, so you can see what happens when you attempt to compile it.

`RAJA::kernel` Execution Policies

This section contains an exercise file `RAJA/exercises/kernelintro-execpols.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/kernelintro-execpols_solution.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make kernelintro-execpols` and `make kernelintro-execpols_solution` from the build directory.

Key RAJA features shown in this section are:

- `RAJA::kernel` kernel execution template and execution policies

The examples in this section illustrate various execution policies for `RAJA::kernel`. The goal is for you to gain an understanding of how execution policies are constructed and used to perform various nested loop execution patterns. All examples use the same simple kernel, which is a three-level loop nest to initialize the entries in an array. The C++ lambda expression representing the kernel inner loop body is identical for all kernel variants described here, whether we are executing the kernel on a CPU sequentially or in parallel with OpenMP, or in parallel on a GPU (CUDA or HIP). The kernels perform the same operations as the examples in the *RAJA::Launch Execution Policies* tutorial section, which uses `RAJA::expt::launch`. By comparing the two sets of examples, you will gain an understanding of the differences between the `RAJA::kernel` and the `RAJA::expt::launch` interfaces.

We begin by defining some constants used throughout the examples and allocating two arrays:

```
//
// 3D tensor has N^3 entries
```

(continues on next page)

(continued from previous page)

```
//
  constexpr int N = 100;
  constexpr int N_tot = N * N * N;
  constexpr double c = 0.0001;
  double* a = memoryManager::allocate<double>(N_tot);
  double* a_ref = memoryManager::allocate<double>(N_tot);
```

Note that we use the 'memory manager' routines contained in the exercise directory to simplify the allocation process. In particular, CUDA unified memory is used when CUDA is enabled to simplify accessing the data on the host or device.

Next, we execute a C-style nested for-loop version of the kernel to initialize the entries in the 'reference' array that we will use to compare the results of other variants for correctness:

```
  for (int k = 0; k < N; ++k ) {
    for (int j = 0; j < N; ++j ) {
      for (int i = 0; i < N; ++i ) {
        a_ref[i+N*(j+N*k)] = c * i * j * k ;
      }
    }
  }
```

Note that we manually compute pointer offsets for the (i,j,k) indices. To simplify the remaining kernel variants we introduce a `RAJA::View` object, which wraps the tensor data pointer and simplifies the multi-dimensional indexing:

```
  RAJA::View< double, RAJA::Layout<3, int> > aView(a, N, N, N);
```

Here 'aView' is a three-dimensional View with extent 'N' in each coordinate based on a three-dimensional `RAJA::Layout` object where the array entries will be accessed using indices of type 'int'. Please see *View and Layout* for more information about the View and Layout types that RAJA provides for various indexing patterns and data layouts.

Using the View, the C-style kernel now looks like:

```
  for (int k = 0; k < N; ++k ) {
    for (int j = 0; j < N; ++j ) {
      for (int i = 0; i < N; ++i ) {
        aView(i, j, k) = c * i * j * k ;
      }
    }
  }
```

Notice how accessing each (i,j,k) entry in the array is more natural, and less error prone, using the View.

The corresponding RAJA sequential version using `RAJA::kernel` is:

```
  using EXEC_POL1 =
    RAJA::KernelPolicy<
      RAJA::statement::For<2, RAJA::loop_exec,     // k
        RAJA::statement::For<1, RAJA::loop_exec,   // j
          RAJA::statement::For<0, RAJA::loop_exec,// i
            RAJA::statement::Lambda<0>
          >
        >
      >
    >;
```

(continues on next page)

(continued from previous page)

```
RAJA::kernel<EXEC_POL1>(
  RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N) ),

  [=]( int i, int j, int k) {
     aView(i, j, k) = c * i * j * k ;
  }
);
```

This should be familiar to the reader who has read the preceding *Basic RAJA::kernel Mechanics and Nested Loop Ordering* section of this tutorial.

Suppose we wanted to parallelize the outer 'k' loop using OpenMP multithreading. A C-style version of this is:

```
#pragma omp parallel for
for (int k = 0; k < N; ++k ) {
  for (int j = 0; j < N; ++j ) {
    for (int i = 0; i < N; ++i ) {
      aView(i, j, k) = c * i * j * k ;
    }
  }
}
```

where we have placed the OpenMP directive `#pragma omp parallel for` before the outer loop of the kernel.

To parallelize all iterations in the entire loop nest, we can apply the OpenMP `collapse(3)` clause to map the iterations for all loop levels to OpenMP threads:

```
#pragma omp parallel for collapse(3)
for (int k = 0; k < N; ++k ) {
  for (int j = 0; j < N; ++j ) {
    for (int i = 0; i < N; ++i ) {
      aView(i, j, k) = c * i * j * k ;
    }
  }
}
```

The corresponding RAJA versions of these two OpenMP variants are, respectively:

```
using EXEC_POL2 =
  RAJA::KernelPolicy<
    RAJA::statement::For<2, RAJA::omp_parallel_for_exec,     // k
      RAJA::statement::For<1, RAJA::loop_exec,               // j
        RAJA::statement::For<0, RAJA::loop_exec,             // i
          RAJA::statement::Lambda<0>
        >
      >
    >
  >;

RAJA::kernel<EXEC_POL2>(
  RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N) ),

  [=]( int i, int j, int k) {
```

(continues on next page)

(continued from previous page)

```
      aView(i, j, k) = c * i * j * k ;
   }
);
```

and

```
using EXEC_POL3 =
  RAJA::KernelPolicy<
    RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec,
                              RAJA::ArgList<2, 1, 0>,  // k, j, i
      RAJA::statement::Lambda<0>
    >
  >;

RAJA::kernel<EXEC_POL3>(
  RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N) ),

  [=]( int i, int j, int k) {
     aView(i, j, k) = c * i * j * k ;
  }
);
```

The first of these, in which we parallelize the outer 'k' loop, replaces the `RAJA::loop_exec` loop execution policy with the `RAJA::omp_parallel_for_exec` policy, which applies the same OpenMP directive to the outer loop used in the C-style variant.

The RAJA OpenMP collapse variant introduces the `RAJA::statement::Collapse` statement type. We use the `RAJA::omp_parallel_collapse_exec` execution policy type and indicate that we want to collapse all three loop levels in the second template argument `RAJA::ArgList<2, 1, 0>`. The integer values in the list indicate the order of the loops in the collapse operation: 'k' (2) outer, 'j' (1) middle, and 'i' (0) inner. The integers represent the order of the lambda arguments and the order of the range segments in the iteration space tuple.

The first RAJA-based kernel for parallel GPU execution using the RAJA CUDA back-end we introduce is:

```
using EXEC_POL5 =
  RAJA::KernelPolicy<
    RAJA::statement::CudaKernel<
      RAJA::statement::For<2, RAJA::cuda_thread_z_loop,      // k
        RAJA::statement::For<1, RAJA::cuda_thread_y_loop,    // j
          RAJA::statement::For<0, RAJA::cuda_thread_x_loop,  // i
            RAJA::statement::Lambda<0>
          >
        >
      >
    >
  >;

RAJA::kernel<EXEC_POL5>(
  RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N) ),

  [=] __device__ ( int i, int j, int k) {
     aView(i, j, k) = c * i * j * k ;
  }
```

(continues on next page)

(continued from previous page)

```
);
```

Here, we use the `RAJA::statement::CudaKernel` statement type to indicate that we want a CUDA kernel to be launched. The 'k', 'j', 'i' iteration variables are mapped to CUDA threads using the CUDA execution policy types `RAJA::cuda_thread_z_loop`, `RAJA::cuda_thread_y_loop`, and `RAJA::cuda_thread_x_loop`, respectively. Thus, we use a a three-dimensional CUDA thread-block to map the loop iterations to CUDA threads. The `_loop` part of each execution policy name indicates that the indexing in the associated portion of the mapping will use a block-stride loop. This is useful to guarantee that the policy will work for any array regardless of size in each coordinate dimension.

To execute the kernel with a prescribed mapping of iterations to a thread-block using RAJA, we could do the following:

```
using EXEC_POL6 =
  RAJA::KernelPolicy<
    RAJA::statement::CudaKernelFixed< i_block_sz * j_block_sz * k_block_sz,
      RAJA::statement::Tile<1, RAJA::tile_fixed<j_block_sz>,
                               RAJA::cuda_block_y_direct,
        RAJA::statement::Tile<0, RAJA::tile_fixed<i_block_sz>,
                                 RAJA::cuda_block_x_direct,
          RAJA::statement::For<2, RAJA::cuda_block_z_direct,       // k
            RAJA::statement::For<1, RAJA::cuda_thread_y_direct,    // j
              RAJA::statement::For<0, RAJA::cuda_thread_x_direct, // i
                RAJA::statement::Lambda<0>
              >
            >
          >
        >
      >
    >
  >;

RAJA::kernel<EXEC_POL6>(
  RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N) ),

  [=] __device__ ( int i, int j, int k) {
     aView(i, j, k) = c * i * j * k ;
  }
);
```

where we have defined the CUDA thread-block dimensions as:

```
constexpr int block_size = 256;
constexpr int i_block_sz = 32;
constexpr int j_block_sz = block_size / i_block_sz;
constexpr int k_block_sz = 1;
```

The `RAJA::statement::CudaKernelFixed` statement indicates that we want to use a fixed thread-block size of 256. To ensure that we are mapping the kernel iterations properly in chunks of 256 threads to each thread-block, we use RAJA tiling statements in which we specify the tile size for each dimension/loop index so that each tile has dimensions (32, 8, 1). For example, the statement `RAJA::statement::Tile<1, RAJA::tile_fixed<j_block_sz>` is used on the 'j' loop, which has a tile size of 8 associated with that dimension. Note that we do not tile the 'k' loop, since the block size is one in that dimension.

The other main difference with the previous block-stride loop kernel version is that we map iterations within each tile directly to threads in a block; for example, using a `RAJA::cuda_block_y_direct` policy type for the 'j'

loop. RAJA *direct* policy types eliminate the block-stride looping, which is not necessary here since we prescribe a block-size of 256 which fits within the thread-block size limitation of the CUDA device programming model.

For context and comparison, here is the same kernel implementation using CUDA directly:

```
  dim3 nthreads_per_block(i_block_sz, j_block_sz, k_block_sz);
  static_assert(i_block_sz*j_block_sz*k_block_sz == block_size,
                "Invalid block_size");

  dim3 nblocks(static_cast<size_t>(RAJA_DIVIDE_CEILING_INT(N, i_block_sz)),
               static_cast<size_t>(RAJA_DIVIDE_CEILING_INT(N, j_block_sz)),
               static_cast<size_t>(RAJA_DIVIDE_CEILING_INT(N, k_block_sz)));

  nested_init<i_block_sz, j_block_sz, k_block_sz>
    <<<nblocks, nthreads_per_block>>>(a, c, N);
  cudaErrchk( cudaGetLastError() );
  cudaErrchk(cudaDeviceSynchronize());
```

The `nested_init` device kernel used here is:

```
template< int i_block_size, int j_block_size, int k_block_size >
__launch_bounds__(i_block_size*j_block_size*k_block_size)
__global__ void nested_init(double* a, double c, int N)
{
  int i = blockIdx.x * i_block_size + threadIdx.x;
  int j = blockIdx.y * j_block_size + threadIdx.y;
  int k = blockIdx.z;

  if ( i < N && j < N && k < N ) {
    a[i+N*(j+N*k)] = c * i * j * k ;
  }
}
```

A few differences between the CUDA and RAJA-CUDA versions are worth noting. First, the CUDA version uses the CUDA `dim3` construct to express the threads-per-block and number of thread-blocks to use: i.e., the `nthreads_per_block` and `nblocks` variable definitions. Note that RAJA provides a macro `RAJA_DIVIDE_CEILING_INT` to perform the proper integer arithmetic to calculate the number of blocks based on the size of the array and the block size in each dimension. Second, the mapping of thread identifiers to the (i,j,k) indices is explicit in the device kernel. Third, an explicit check of the (i,j,k) values is required in the CUDA implementation to avoid addressing memory out-of-bounds; i.e., `if ( i < N && j < N && k < N ) ...`. The RAJA kernel variants set similar definitions internally and **mask out indices that would be out-of-bounds.** Note that we also inserted additional error checking with `static_assert` and `cudaErrchk`, which is a RAJA macro, for printing CUDA device error codes, to catch device errors if there are any.

Lastly, we show the RAJA HIP variants of the kernel, which are semantically identical to the RAJA CUDA variants we just described. First, the RAJA-HIP block-stride loop variant:

```
  using EXEC_POL7 =
    RAJA::KernelPolicy<
      RAJA::statement::HipKernel<
        RAJA::statement::For<2, RAJA::hip_thread_z_loop,      // k
          RAJA::statement::For<1, RAJA::hip_thread_y_loop,    // j
            RAJA::statement::For<0, RAJA::hip_thread_x_loop,  // i
              RAJA::statement::Lambda<0>
            >
          >
        >
      >
```

(continues on next page)

(continued from previous page)

```
  >;

RAJA::kernel<EXEC_POL7>(
  RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N),
                    RAJA::TypedRangeSegment<int>(0, N) ),

  [=] __device__ ( int i, int j, int k) {
     d_aView(i, j, k) = c * i * j * k ;
  }
);
```

and then the RAJA-HIP fixed thread-block size, tiled, direct thread mapping version:

```
using EXEC_POL8 =
  RAJA::KernelPolicy<
    RAJA::statement::HipKernelFixed< i_block_sz * j_block_sz * k_block_sz,
      RAJA::statement::Tile<1, RAJA::tile_fixed<j_block_sz>,
                               RAJA::hip_block_y_direct,
        RAJA::statement::Tile<0, RAJA::tile_fixed<i_block_sz>,
                                 RAJA::hip_block_x_direct,
          RAJA::statement::For<2, RAJA::hip_block_z_direct,       // k
            RAJA::statement::For<1, RAJA::hip_thread_y_direct,    // j
              RAJA::statement::For<0, RAJA::hip_thread_x_direct, // i
                RAJA::statement::Lambda<0>
              >
            >
          >
        >
      >
    >
  >;

RAJA::kernel<EXEC_POL8>(
   RAJA::make_tuple( RAJA::TypedRangeSegment<int>(0, N),
                     RAJA::TypedRangeSegment<int>(0, N),
                     RAJA::TypedRangeSegment<int>(0, N) ),

  [=] __device__ ( int i, int j, int k) {
     d_aView(i, j, k) = c * i * j * k ;
  }
);
```

The only differences are that type names are changed to replace 'CUDA' types with 'HIP' types to use the RAJA HIP back-end.

OffsetLayout: Five-point Stencil

This section contains an exercise file `RAJA/exercises/offset-layout-stencil.cpp` for you to work through if you wish to get some practice with RAJA. The file `RAJA/exercises/offset-layout-stencil.cpp` contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute `make offset-layout-stencil` and `make offset-layout-stencil_solution` from the build directory.

Key RAJA features shown in the following example:

- `RAJA::kernel` loop execution template and execution policies
- `RAJA::View` multi-dimensional data access
- `RAJA:make_offset_layout` method to create an offset Layout

The examples in this section apply a five-point stencil to the interior cells of a two-dimensional lattice and store a resulting sum in a second lattice of equal size. The five-point stencil associated with a lattice cell accumulates the value in the cell and each of its four neighbors. We use `RAJA::View` and `RAJA::OffsetLayout` constructs to simplify the multi-dimensional indexing so that we can write the stencil operation naturally, as such:

```
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 $[-1, N_r + 1) \times [-1, N_c + 1)$ is the full lattice index range.

For reference and comparison to the `RAJA::kernel` implementations described below, we begin by walking through a C-style version of the stencil computation. First, we define the size of our lattice:

```
//
// Define num of interior cells in row/cols in a lattice
//
  constexpr int N_r = 5;
  constexpr int N_c = 4;

//
// Define total num of cells in rows/cols in a lattice
```

(continues on next page)

(continued from previous page)

```
//
  constexpr int totCellsInRow = N_r + 2;
  constexpr int totCellsInCol = N_c + 2;

//
// Define total num of cells in a lattice
//
  constexpr int totCells = totCellsInRow * totCellsInCol;
```

Then, after allocating input and output arrays, we initialize the input:

```
  for (int row = 1; row <= N_r; ++row) {
    for (int col = 1; col <= N_c; ++col) {
      int id = col + totCellsInCol * row;
      input[id] = 1;
    }
  }
```

and compute the reference output solution:

```
  for (int row = 1; row <= N_r; ++row) {
    for (int col = 1; col <= N_c; ++col) {

      int id = col + totCellsInCol * row;
      output_ref[id] = input[id] + input[id + 1]
                       + input[id - 1]
                       + input[id + totCellsInCol]
                       + input[id - totCellsInCol];
    }
  }
```

RAJA Offset Layouts

We use the `RAJA::make_offset_layout` method to construct a `RAJA::OffsetLayout` object that we use to create `RAJA::View` objects for our input and output data arrays:

```
  const int DIM = 2;

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

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

Here, the row index range is $[-1, N_r + 1)$, and the column index range is $[-1, N_c + 1)$. The first argument to each call to the `RAJA::View` constructor is the pointer to the array that holds the View data. The second argument is the `RAJA::OffsetLayout` object.

`RAJA::OffsetLayout` objects allow us to write loops over data arrays using non-zero based indexing and without having to manually compute offsets into the arrays.

For more information about RAJA View and Layout types, please see *View and Layout*.

RAJA Kernel Variants

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

```
RAJA::TypedRangeSegment<int> col_range(0, N_c);
RAJA::TypedRangeSegment<int> row_range(0, N_r);
```

Now, we have all the ingredients to implement the stencil computation using `RAJA::kernel`. Here is a sequential CPU variant:

```
using NESTED_EXEC_POL1 =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, RAJA::loop_exec,     // row
      RAJA::statement::For<0, RAJA::loop_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);

                               });
```

This RAJA variant does the computation as the C-style variant introduced above.

Since the input and output arrays are distinct, the stencil computation is data parallel. Thus, we can use `RAJA::kernel` and an appropriate execution policy to run the computation in parallel. Here is an OpenMP collapse variant that maps the row-column product index space to OpenMP threads:

```
using NESTED_EXEC_POL2 =
  RAJA::KernelPolicy<
    RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec,
                              RAJA::ArgList<1, 0>,    // row, col
      RAJA::statement::Lambda<0>
    >
  >;

RAJA::kernel<NESTED_EXEC_POL2>(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);

                               });
```

Note that the lambda expression representing the kernel body is identical to the `RAJA::kernel` sequential version.

Here are variants for CUDA

```
using NESTED_EXEC_POL3 =
  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::kernel<NESTED_EXEC_POL3>(RAJA::make_tuple(col_range, row_range),
                               [=] RAJA_DEVICE(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);

                               });
```

and HIP

```
using NESTED_EXEC_POL4 =
  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::kernel<NESTED_EXEC_POL4>(RAJA::make_tuple(col_range, row_range),
                               [=] RAJA_DEVICE(int col, int row) {

                                 d_outputView(row, col) =
                                       d_inputView(row, col)
                                     + d_inputView(row - 1, col)
                                     + d_inputView(row + 1, col)
                                     + d_inputView(row, col - 1)
                                     + d_inputView(row, col + 1);
                               });
```

The only difference between the CPU and GPU variants is that the RAJA macro `RAJA_DEVICE` is used to decorate the lambda expression with the `__device__` annotation, which is required when capturing a lambda for use in a GPU device environment as we have discussed in other examples in this tutorial.

One other point to note is that the CUDA variant in the exercise files uses Unified Memory and the HIP variant uses distinct host and device memory arrays, with explicit host-device data copy operations. Thus, new `RAJA::View` objects were created for the HIP variant to wrap the device data pointers used in the HIP kernel. Please see the exercise files for this example for details.

Nested Loops with `RAJA::expt::launch`

The examples in this section illustrate how to use `RAJA::expt::launch` to create an run time selectable execution space for expressing algorithms as nested loops.

`RAJA::Launch` Basics

There are no exercise files to work through for this section. Instead, there is an example source file `RAJA/examples/tut_launch_basic.cpp` which contains complete code examples of the concepts described here.

Key RAJA features shown in the following examples are:

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

In this example, we introduce the RAJA Launch framework and discuss hierarchical loop-based parallelism. Kernel execution details with RAJA Launch occur inside the lambda expression passed to the `RAJA::launch` method, which defines an execution space:

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

  /* Kernel code goes here */

});
```

The `RAJA::launch` method accepts a `RAJA::LaunchPolicy` template parameter that can be defined using up to two policies (host and device). For example, the following creates an execution space for a sequential and CUDA kernel dispatch:

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

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

Inside the execution space, developers write a kernel using nested `RAJA::loop` methods. The manner in which each loop is executed is determined by a template parameter type, which indicates how the corresponding iterates are mapped to the Teams/Threads configuration defined by the `RAJA::LaunchParams` type passed as the second argument to the `RAJA::launch` method. Following the CUDA and HIP programming models, this defines an hierarchical structure in which outer loops are executed by thread-teams and inner loops are executed by threads in a team.

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

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

(continues on next page)

(continued from previous page)

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


            });
          });

        });
      });
```

The mapping between teams and threads to the underlying programming model depends on how the `RAJA::loop` template parameter types are defined. For example, we may define host and device mapping strategies as:

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

Here, the `RAJA::LoopPolicy` type holds both the host (CPU) and device (CUDA GPU) 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 (or HIP) thread blocks, while the `thread_x/y` policies will map loop iterations directly to threads in a CUDA (or HIP) thread block. The direct CUDA equivalent of the kernel body using the policy shown above is:

```
  {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);

        }
      }

    }
  }
```

RAJA::Launch Execution Policies

This section contains an exercise file RAJA/exercises/launchintro-execpols.cpp for you to work through if you wish to get some practice with RAJA. The file RAJA/exercises/launchintro-execpols_solution.cpp contains complete working code for the examples discussed in this section. You can use the solution file to check your work and for guidance if you get stuck. To build the exercises execute make launchintro-execpols and make launchintro-execpols_solution from the build directory.

Key RAJA features shown in this section are:

- RAJA::launch kernel execution environment template
- RAJA::loop loop execution template and execution policies

The examples in this section illustrate how to construct nested loop kernels inside an RAJA::launch execution environment. In particular, the goal is for you to gain an understanding of how to use execution policies with nested RAJA::loop method calls to perform various nested loop execution patterns. All examples use the same simple kernel, which is a three-level loop nest to initialize the entries in an array. The kernels perform the same operations as the examples in *RAJA::kernel Execution Policies*. By comparing the two sets of examples, you will gain an understanding of the differences between the RAJA::kernel and the RAJA::launch interfaces.

We begin by defining some constants used throughout the examples and allocating arrays to represent the array data:

```
//
// 3D tensor has N^3 entries
//
  constexpr int N = 100;
  constexpr int N_tot = N * N * N;
  constexpr double c = 0.0001;
  double* a = memoryManager::allocate<double>(N_tot);
  double* a_ref = memoryManager::allocate<double>(N_tot);
```

Note that we use the 'memory manager' routines contained in the exercise directory to simplify the allocation process. In particular, CUDA unified memory is used when CUDA is enabled to simplify accessing the data on the host or device.

Next, we execute a C-style nested for-loop version of the kernel to initialize the entries in the 'reference' array that we will use to compare the results of other variants for correctness:

```
  for (int k = 0; k < N; ++k ) {
    for (int j = 0; j < N; ++j ) {
      for (int i = 0; i < N; ++i ) {
        a_ref[i+N*(j+N*k)] = c * i * j * k ;
      }
    }
  }
```

Note that we manually compute the pointer offsets for the (i,j,k) indices. To simplify the remaining kernel variants we introduce a RAJA::View object, which wraps the array data pointer and simplifies the multi-dimensional indexing:

```
  RAJA::View< double, RAJA::Layout<3, int> > aView(a, N, N, N);
```

Here 'aView' is a three-dimensional View with extent 'N' in each coordinate based on a three-dimensional RAJA::Layout object where the array entries will be accessed using indices of type 'int'. indices of type int. Please see *View and Layout* for more information about the View and Layout types that RAJA provides for various indexing patterns and data layouts.

Using the View, the C-style kernel looks like:

```
  for (int k = 0; k < N; ++k ) {
    for (int j = 0; j < N; ++j ) {
      for (int i = 0; i < N; ++i ) {
        aView(i, j, k) = c * i * j * k ;
      }
    }
  }
```

Notice how accessing each (i,j,k) entry in the array is more natural, and less error prone, using the View.

The corresponding RAJA sequential version using `RAJA::launch` is:

```
  using loop_policy_1 = RAJA::LoopPolicy<RAJA::loop_exec>;
  using launch_policy_1 = RAJA::LaunchPolicy<RAJA::seq_launch_t>;

  RAJA::launch<launch_policy_1>
    (RAJA::LaunchParams(), //LaunchParams may be empty when running on the host
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::loop<loop_policy_1>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int k) {
         RAJA::loop<loop_policy_1>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int␣
↪j) {
            RAJA::loop<loop_policy_1>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&]␣
↪(int i) {

                aView(i, j, k) = c * i * j * k ;

            });
         });
      });
  });
```

This should be familiar to the reader who has read through the preceding *RAJA::Launch Basics* section of this tutorial. As the `RAJA::launch` method is templated on a host execution policy, the `RAJA::LaunchParams` object can be defined without arguments as loop methods will get dispatched as standard C-Style for-loops.

Suppose we wanted to parallelize the outer 'k' loop using OpenMP multithreading. A C-style version of this is:

```
  #pragma omp parallel for
  for (int k = 0; k < N; ++k ) {
    for (int j = 0; j < N; ++j ) {
      for (int i = 0; i < N; ++i ) {
        aView(i, j, k) = c * i * j * k ;
      }
    }
  }
```

where we have placed the OpenMP directive `#pragma omp parallel for` before the outer loop of the kernel.

The corresponding RAJA versions of the C-style OpenMP variant is:

```
  using omp_policy_2 = RAJA::LoopPolicy<RAJA::omp_for_exec>;
  using loop_policy_2 = RAJA::LoopPolicy<RAJA::loop_exec>;
  using launch_policy_2 = RAJA::LaunchPolicy<RAJA::omp_launch_t>;

  RAJA::launch<launch_policy_2>
    (RAJA::LaunchParams(), //LaunchParams may be empty when running on the host
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {
```

(continues on next page)

(continued from previous page)

```
     RAJA::loop<omp_policy_2>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int k) {
        RAJA::loop<loop_policy_2>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int␣
↪j) {
           RAJA::loop<loop_policy_2>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&]␣
↪(int i) {

               aView(i, j, k) = c * i * j * k ;

           });
        });
     });
  });
```

With the OpenMP version above, `RAJA::launch` method is templated on a `RAJA::omp_launch_t` execution policy. The policy is used to create an OpenMP parallel region, loop iterations may then be distributed using `RAJA::loop` methods templated on `RAJA::omp_for_exec` execution policies. As before, the `RAJA::LaunchParams` object may be initialized without grid dimensions as the CPU does not require specifying a compute grid.

The first RAJA-based kernel for parallel GPU execution using the RAJA CUDA back-end we introduce is:

```
  using cuda_teams_z_3 = RAJA::LoopPolicy<RAJA::cuda_block_z_direct>;
  using cuda_global_thread_y_3 = RAJA::LoopPolicy<RAJA::cuda_global_thread_y>;
  using cuda_global_thread_x_3 = RAJA::LoopPolicy<RAJA::cuda_global_thread_x>;

  const bool async_3 = false;
  using launch_policy_3 = RAJA::LaunchPolicy<RAJA::cuda_launch_t<async_3>>;

  RAJA::launch<launch_policy_3>
    (RAJA::LaunchParams(RAJA::Teams(n_blocks_i ,n_blocks_j, n_blocks_k),
                        RAJA::Threads(i_block_sz, j_block_sz, k_block_sz)),
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::loop<cuda_teams_z_3>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int k)
↪{
        RAJA::loop<cuda_global_thread_y_3>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&
↪] (int j) {
          RAJA::loop<cuda_global_thread_x_3>(ctx, RAJA::TypedRangeSegment<int>(0, N),␣
↪[&] (int i) {

            aView(i, j, k) = c * i * j * k ;

          });
        });
      });
  });
```

where we have defined the CUDA thread-block dimensions as:

```
  constexpr int block_size = 256;
  constexpr int i_block_sz = 32;
  constexpr int j_block_sz = block_size / i_block_sz;
  constexpr int k_block_sz = 1;

  const int n_blocks_i = RAJA_DIVIDE_CEILING_INT(N,i_block_sz);
  const int n_blocks_j = RAJA_DIVIDE_CEILING_INT(N,j_block_sz);
```

(continues on next page)

(continued from previous page)

```
  const int n_blocks_k = RAJA_DIVIDE_CEILING_INT(N,k_block_sz);
```

Here, we use the `RAJA::cuda_launch_t` policy type to indicate that we want a CUDA kernel to be launched. The 'k', 'j', 'i' iteration variables are mapped to CUDA threads and blocks using the CUDA execution policy types `RAJA::cuda_block_z_direct`, `RAJA::cuda_global_thread_y`, and `RAJA::cuda_global_thread_x`, respectively. Thus, we use a two-dimensional CUDA thread-block and three-dimensional compute grid to map the loop iterations to CUDA threads. In comparison to the RAJA CUDA example in *RAJA::kernel Execution Policies* , `RAJA::loop` methods support execution policies, which enable mapping directly to the global thread ID of a compute grid.

Using a combination of `RAJA::tile` and `RAJA::loop` methods, we can create a loop tiling platform portable implementation. Here, is a CUDA variant:

```
  using cuda_teams_z_4 = RAJA::LoopPolicy<RAJA::cuda_block_z_direct>;
  using cuda_teams_y_4 = RAJA::LoopPolicy<RAJA::cuda_block_y_direct>;
  using cuda_teams_x_4 = RAJA::LoopPolicy<RAJA::cuda_block_x_direct>;

  using cuda_threads_y_4 = RAJA::LoopPolicy<RAJA::cuda_thread_y_direct>;
  using cuda_threads_x_4 = RAJA::LoopPolicy<RAJA::cuda_thread_x_direct>;

  const bool async_4 = false;
  using launch_policy_4 = RAJA::LaunchPolicy<RAJA::cuda_launch_t<async_4>>;

  RAJA::launch<launch_policy_4>
    (RAJA::LaunchParams(RAJA::Teams(n_blocks_i, n_blocks_j, n_blocks_k),
                        RAJA::Threads(i_block_sz, j_block_sz, k_block_sz)),
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::loop<cuda_teams_z_4>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int k)
↪{

        RAJA::tile<cuda_teams_y_4>
          (ctx, j_block_sz, RAJA::TypedRangeSegment<int>(0, N), [&]␣
↪(RAJA::TypedRangeSegment<int> const &j_tile) {

          RAJA::tile<cuda_teams_x_4>
            (ctx, i_block_sz, RAJA::TypedRangeSegment<int>(0, N), [&]␣
↪(RAJA::TypedRangeSegment<int> const &i_tile) {

            RAJA::loop<cuda_threads_y_4>(ctx, j_tile, [&] (int j) {
                RAJA::loop<cuda_threads_x_4>(ctx, i_tile, [&] (int i) {

                    aView(i, j, k) = c * i * j * k ;

                  });
              });

            });
          });

      });
    });
```

We consider the kernel to be portable, because all of the execution policy types and execution parameters can be replaced by other types and values without changing the kernel code directly.

The `RAJA::tile` methods are used to partition an iteration space into tiles to be used within a `RAJA::loop`

method. The '{i,j,k}_block_sz' arguments passed to the RAJA::tile function specify the tile size for each loop. In the case of GPU programming models, we define the tile size to correspond to the number of threads in a given dimension. Execution tile and loop execution policies are chosen to have CUDA blocks and threads map directly to tiles and entries in a tile.

For context and comparison, here is the same kernel implementation using CUDA directly:

```
  dim3 nthreads_per_block(i_block_sz, j_block_sz, k_block_sz);
  static_assert(i_block_sz*j_block_sz*k_block_sz == block_size,
                "Invalid block_size");

  dim3 nblocks(static_cast<size_t>(RAJA_DIVIDE_CEILING_INT(N, i_block_sz)),
               static_cast<size_t>(RAJA_DIVIDE_CEILING_INT(N, j_block_sz)),
               static_cast<size_t>(RAJA_DIVIDE_CEILING_INT(N, k_block_sz)));

  nested_init<i_block_sz, j_block_sz, k_block_sz>
    <<<nblocks, nthreads_per_block>>>(a, c, N);
  cudaErrchk( cudaGetLastError() );
  cudaErrchk(cudaDeviceSynchronize());
```

The nested_init device kernel used here is:

```
template< int i_block_size, int j_block_size, int k_block_size >
__launch_bounds__(i_block_size*j_block_size*k_block_size)
__global__ void nested_init(double* a, double c, int N)
{
  int i = blockIdx.x * i_block_size + threadIdx.x;
  int j = blockIdx.y * j_block_size + threadIdx.y;
  int k = blockIdx.z;

  if ( i < N && j < N && k < N ) {
    a[i+N*(j+N*k)] = c * i * j * k ;
  }
}
```

A few differences between the CUDA and RAJA-CUDA versions are worth noting. First, the CUDA version uses the CUDA dim3 construct to express the threads-per-block and number of thread-blocks to use: i.e., the nthreads_per_block and nblocks variable definitions. The RAJA::launch interface takes compute dimensions through a RAJA::LaunchParams object. RAJA provides a macro RAJA_DIVIDE_CEILING_INT to perform the proper integer arithmetic to calculate the number of blocks based on the size of the array and the block size in each dimension. Second, the mapping of thread identifiers to the (i,j,k) indices is explicit in the device kernel. Third, an explicit check of the (i,j,k) values is required in the CUDA implementation to avoid addressing memory out-of-bounds; i.e., `if ( i < N && j < N && k < N ) ...`. The RAJA variants set similar definitions internally and **mask out indices that would be out-of-bounds.** Note that we also inserted additional error checking with static_assert and cudaErrchk, which is a RAJA macro, for printing CUDA device error codes, to catch device errors if there are any.

Lastly, we show the RAJA HIP variants of the kernel, which are semantically identical to the RAJA CUDA variants. First, the RAJA-HIP global-thread variant:

```
  using hip_teams_z_5 = RAJA::LoopPolicy<RAJA::hip_block_z_direct>;
  using hip_global_thread_y_5 = RAJA::LoopPolicy<RAJA::hip_global_thread_y>;
  using hip_global_thread_x_5 = RAJA::LoopPolicy<RAJA::hip_global_thread_x>;

  const bool async_5 = false;
  using launch_policy_5 = RAJA::LaunchPolicy<RAJA::hip_launch_t<async_5>>;
```

(continues on next page)

(continued from previous page)

```
  RAJA::launch<launch_policy_5>
    (RAJA::LaunchParams(RAJA::Teams(n_blocks_i, n_blocks_j, n_blocks_k),
                        RAJA::Threads(i_block_sz, j_block_sz, k_block_sz)),
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

       RAJA::loop<hip_teams_z_5>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int k)
↪{
           RAJA::loop<hip_global_thread_y_5>(ctx, RAJA::TypedRangeSegment<int>(0, N),␣
↪[&] (int j) {
               RAJA::loop<hip_global_thread_x_5>(ctx, RAJA::TypedRangeSegment<int>(0,␣
↪N), [&] (int i) {

                   d_aView(i, j, k) = c * i * j * k ;

           });
         });
       });

  });
```

and then the RAJA Launch HIP fixed thread-block size, tiled, direct thread mapping version:

```
  using hip_teams_z_6 = RAJA::LoopPolicy<RAJA::hip_block_z_direct>;
  using hip_teams_y_6 = RAJA::LoopPolicy<RAJA::hip_block_y_direct>;
  using hip_teams_x_6 = RAJA::LoopPolicy<RAJA::hip_block_x_direct>;

  using hip_threads_y_6 = RAJA::LoopPolicy<RAJA::hip_thread_y_direct>;
  using hip_threads_x_6 = RAJA::LoopPolicy<RAJA::hip_thread_x_direct>;

  const bool async_6 = false;
  using launch_policy_6 = RAJA::LaunchPolicy<RAJA::hip_launch_t<async_6>>;

  RAJA::launch<launch_policy_6>
    (RAJA::LaunchParams(RAJA::Teams(n_blocks_i, n_blocks_j, n_blocks_k),
                        RAJA::Threads(i_block_sz, j_block_sz, k_block_sz)),
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::loop<hip_teams_z_6>(ctx, RAJA::TypedRangeSegment<int>(0, N), [&] (int k) {

        RAJA::tile<hip_teams_y_6>
          (ctx, j_block_sz, RAJA::TypedRangeSegment<int>(0, N), [&]␣
↪(RAJA::TypedRangeSegment<int> const &j_tile) {

          RAJA::tile<hip_teams_x_6>
            (ctx, i_block_sz, RAJA::TypedRangeSegment<int>(0, N), [&]␣
↪(RAJA::TypedRangeSegment<int> const &i_tile) {

            RAJA::loop<hip_threads_y_6>(ctx, j_tile, [&] (int j) {
                RAJA::loop<hip_threads_x_6>(ctx, i_tile, [&] (int i) {

                    d_aView(i, j, k) = c * i * j * k ;

                });
              });

            });
          });
```

(continues on next page)

(continued from previous page)

```
      });
    });
```

The only differences are that type names are changed to replace 'CUDA' types with 'HIP' types to use the RAJA HIP back-end.

Naming kernels for NVTX/ROCTX tools

There are no exercise files to work through for this section. Instead, there is an example source file `RAJA/examples/teams_reductions.cpp` which contains complete code examples of the concepts described here.

Key RAJA feature shown in the following example:

- Naming kernels using an optional argument in `RAJA::launch` methods.

In this example, we illustrate kernel naming capabilities within the RAJA Launch framework for use with NVTX or ROCTX region naming capabilities.

To name a `RAJA::launch` kernel, a string name is passed as an argument before the lambda

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

    /* Kernel body code goes 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, nvprof kernels are identified as ranges of GPU activity using the provided kernel 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
```

Similarly, ROCm tools can be used to generate traces of a profile and the resulting json file can be viewed using tools such as Perfetto.

In future work, we plan to add support to other profiling tools. Thus, API changes may occur based on user feedback and integration with other tools. Enabling NVTX profiling with RAJA Launch requires RAJA to be configured with RAJA_ENABLE_NV_TOOLS_EXT=ON. or RAJA_ENABLE_ROCTX=ON for ROCTX profiling on AMD platforms platforms.

Comparing RAJA::kernel and RAJA::expt::launch: Matrix-Transpose

In this section, we compare RAJA::kernel and RAJA::expt::launch implementations of a matrix transpose algorithm. We illustrate implementation differences of the two interfaces as we build upon each example with more complex features.

Matrix Transpose

In *RAJA::kernel Execution Policies* and *RAJA::Launch Execution Policies*, we presented a simple array initialization kernel using RAJA::kernel and RAJA::launch interfaces, respectively, and compared the two. This section describes the implementation of a matrix transpose kernel using both RAJA::kernel and RAJA::launch interfaces. The intent is to compare and contrast the two, as well as introduce additional features of the interfaces.

There are exercise files RAJA/exercises/kernel-matrix-transpose.cpp and RAJA/exercises/launch-matrix-transpose.cpp for you to work through if you wish to get some practice with RAJA. The files RAJA/exercises/kernel-matrix-transpose_solution.cpp and RAJA/exercises/launch-matrix-transpose_solution.cpp contain complete working code for the examples. You can use the solution files to check your work and for guidance if you get stuck. To build the exercises execute make (kernel/launch)-matrix-transpose and make (kernel/launch)-matrix-transpose_solution from the build directory.

Key RAJA features shown in this example are:

- RAJA::kernel method and kernel execution policies
- RAJA::launch method and kernel execution interface

In the 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 $N_c \times N_r$.

First we define our matrix dimensions

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

and wrap the data pointers for the matrices 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, a C-style for-loop implementation looks like this:

```
for (int row = 0; row < N_r; ++row) {
  for (int col = 0; col < N_c; ++col) {
      Atview(col, row) = Aview(row, col);
  }
}
```

RAJA::kernel Implementation

For RAJA::kernel variants, we use RAJA::statement::For and RAJA::statement::Lambda statement types in the execution policies. The complete sequential RAJA::kernel variant is:

```
using KERNEL_EXEC_POL =
  RAJA::KernelPolicy<
```

(continues on next page)

(continued from previous page)

```
    RAJA::statement::For<1, RAJA::loop_exec,
      RAJA::statement::For<0, RAJA::loop_exec,
        RAJA::statement::Lambda<0>
       >
    >
  >;

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

A CUDA `RAJA::kernel` variant for the GPU is similar with different policies in the `RAJA::statement::For` statements:

```
using KERNEL_EXEC_POL_CUDA =
  RAJA::KernelPolicy<
    RAJA::statement::CudaKernel<
      RAJA::statement::For<1, RAJA::cuda_thread_x_loop,
        RAJA::statement::For<0, RAJA::cuda_thread_y_loop,
                             RAJA::statement::Lambda<0>
        >
      >
    >
  >;

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

A notable difference between the CPU and GPU execution policy is the insertion of the `RAJA::statement::CudaKernel` type in the GPU version, which indicates that the execution will launch a CUDA device kernel.

In the CUDA `RAJA::kernel` variant above, the thread-block size and and number of blocks to launch is determined by the implementation of the `RAJA::kernel` execution policy constructs using the sizes of the `RAJA::TypedRangeSegment` objects in the iteration space tuple.

`RAJA::launch` Implementation

For `RAJA::launch` variants, we use `RAJA::loop` methods to write a loop hierarchy within the kernel execution space. For a sequential implementation, we pass the `RAJA::seq_launch_t` template parameter to the launch method and pass the `RAJA::loop_exec` parameter to the loop methods. The complete sequential `RAJA::launch` variant is:

```
using loop_policy_seq = RAJA::LoopPolicy<RAJA::loop_exec>;
using launch_policy_seq = RAJA::LaunchPolicy<RAJA::seq_launch_t>;

RAJA::launch<launch_policy_seq>(
  RAJA::LaunchParams(), //LaunchParams may be empty when running on the host
  [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

    RAJA::loop<loop_policy_seq>(ctx, row_Range, [&] (int row) {
      RAJA::loop<loop_policy_seq>(ctx, col_Range, [&] (int col) {
```

(continues on next page)

(continued from previous page)

```
          Atview(col, row) = Aview(row, col);

      });
    });

});
```

A CUDA `RAJA::launch` variant for the GPU is similar with CUDA policies in the `RAJA::loop` methods. The complete `RAJA::launch` variant is:

```
using cuda_thread_x = RAJA::LoopPolicy<RAJA::cuda_thread_x_loop>;
using cuda_thread_y = RAJA::LoopPolicy<RAJA::cuda_thread_y_loop>;

const bool async = false; //execute asynchronously
using launch_policy_cuda = RAJA::LaunchPolicy<RAJA::cuda_launch_t<async>>;

RAJA::launch<launch_policy_cuda>
  (RAJA::LaunchParams(RAJA::Teams(1), RAJA::Threads(16,16)),
  [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

    RAJA::loop<cuda_thread_y>(ctx, row_Range, [&] (int row) {
      RAJA::loop<cuda_thread_x>(ctx, col_Range, [&] (int col) {

          Atview(col, row) = Aview(row, col);

      });
    });

});
```

A notable difference between the CPU and GPU `RAJA::launch` implementations is the definition of the compute grid. For the CPU version, the argument list is empty for the `RAJA::LaunchParams` constructor. For the CUDA GPU implementation, we define a 'Team' of one two-dimensional thread-block with 16 x 16 = 256 threads.

Tiled Matrix Transpose

This section describes the implementation of a tiled matrix transpose kernel using both `RAJA::kernel` and `RAJA::launch` interfaces. The intent is to compare and contrast the two. The discussion builds on *Matrix Transpose* by adding tiling to the matrix transpose implementation.

There are exercise files `RAJA/exercises/kernel-matrix-transpose-tiled.cpp` and `RAJA/exercises/launch-matrix-transpose-tiled.cpp` for you to work through if you wish to get some practice with RAJA. The files `RAJA/exercises/kernel-matrix-transpose-tiled_solution.cpp` and `RAJA/exercises/launch-matrix-transpose-tiled_solution.cpp` contain complete working code for the examples. You can use the solution files to check your work and for guidance if you get stuck. To build the exercises execute `make (kernel/launch)-matrix-transpose-tiled` and `make (kernel/launch)-matrix-transpose-tiled_solution` from the build directory.

Key RAJA features shown in this example are:

- `RAJA::kernel` method and execution policies, and the `RAJA::statement::Tile` type
- `RAJA::launch` method and execution policies, and the `RAJA::tile` type

As in *Matrix Transpose*, we compute the transpose of an input matrix A of size $N_r \times N_c$ and storing the result in a second matrix At of size $N_c \times N_r$.

We will compute the matrix transpose using a tiling algorithm, which iterates over tiles and transposes the matrix entries in each tile. The algorithm involves outer and inner loops to iterate over the tiles and matrix entries within each tile, respectively.

As in *Matrix Transpose*, we start by defining the matrix dimensions. Additionally, we define a tile size smaller than the matrix dimensions and determine the number of tiles in each dimension. 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.

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

constexpr int TILE_DIM = 16;

constexpr int outer_Dimc = (N_c - 1) / TILE_DIM + 1;
constexpr 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);
```

The C-style for-loop 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: To prevent indexing out of bounds, when the tile dimensions do not divide evenly the matrix dimensions, the algorithm requires a bounds check in the inner loops.

`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 TILED_KERNEL_EXEC_POL =
    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::For<1, RAJA::loop_exec,
            RAJA::statement::For<0, RAJA::loop_exec,
              RAJA::statement::Lambda<0>
            >
          >
        >
      >
    >;

  RAJA::kernel<TILED_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 explicit bounds checking logic in the C-style variant is not needed. Note that the integer template parameters in the `RAJA::statement::For` types refer to the entries in the iteration space tuple passed to the `RAJA::kernel` method.

The `RAJA::kernel` CUDA variant is similar with sequential policies replaced with CUDA execution policies:

```
  using TILED_KERNEL_EXEC_POL_CUDA =
    RAJA::KernelPolicy<
      RAJA::statement::CudaKernel<
        RAJA::statement::Tile<1, RAJA::tile_fixed<TILE_DIM>, RAJA::cuda_block_y_loop,
          RAJA::statement::Tile<0, RAJA::tile_fixed<TILE_DIM>, RAJA::cuda_block_x_
↪loop,
            RAJA::statement::For<1, RAJA::cuda_thread_x_direct,
              RAJA::statement::For<0, RAJA::cuda_thread_y_direct,
                RAJA::statement::Lambda<0>
              >
            >
          >
        >
      >
    >;

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

A notable difference between the CPU and GPU execution policy is the insertion of the `RAJA::statement::CudaKernel` type in the GPU version, which indicates that the execution will launch a CUDA device kernel.

The CUDA thread-block dimensions are set based on the tile dimensions and the iterates withing each tile are mapped directly to GPU threads in each block due to the `RAJA::cuda_thread_{x, y}_direct` policies.

RAJA::launch Variants

For RAJA::launch variants, we use RAJA::tile methods for the outer loop tiling and RAJA::loop methods to iterate within the tiles. The complete sequential tiled RAJA::launch variant is:

```
  using loop_pol_1 = RAJA::LoopPolicy<RAJA::loop_exec>;
  using launch_policy_1 = RAJA::LaunchPolicy<RAJA::seq_launch_t>;

  RAJA::launch<launch_policy_1>(RAJA::LaunchParams(), //LaunchParams may be empty␣
↪when running on the cpu
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::tile<loop_pol_1>(ctx, TILE_DIM, row_Range, [&] (RAJA::TypedRangeSegment
↪<int> const &row_tile) {

        RAJA::tile<loop_pol_1>(ctx, TILE_DIM, col_Range, [&] (RAJA::TypedRangeSegment
↪<int> const &col_tile) {

          RAJA::loop<loop_pol_1>(ctx, row_tile, [&] (int row) {
            RAJA::loop<loop_pol_1>(ctx, col_tile, [&] (int col) {

              Atview(col, row) = Aview(row, col);

            });
          });

        });
      });

  });
```

Similar to the RAJA::statement::Tile type in the RAJA::kernel variant above, the RAJA::tile method computes the number of tiles needed to iterate over all matrix entries in each dimension and generates a corresponding iteration space for each tile, which is used to generate loops for the inner RAJA::loop methods. Thus, the explicit bounds checking logic in the C-style variant is not needed.

A CUDA RAJA::launch tiled variant for the GPU is similar with CUDA policies in the RAJA::loop methods. The complete RAJA::launch variant is:

```
  using cuda_teams_y = RAJA::LoopPolicy<RAJA::cuda_block_y_direct>;
  using cuda_teams_x = RAJA::LoopPolicy<RAJA::cuda_block_x_direct>;

  using cuda_threads_y = RAJA::LoopPolicy<RAJA::cuda_thread_y_direct>;
  using cuda_threads_x = RAJA::LoopPolicy<RAJA::cuda_thread_x_direct>;

  const bool cuda_async = false;
  using cuda_launch_policy = RAJA::LaunchPolicy<RAJA::cuda_launch_t<cuda_async>>;

  RAJA::launch<cuda_launch_policy>(
    RAJA::LaunchParams(RAJA::Teams(n_blocks_c, n_blocks_r),
                       RAJA::Threads(c_block_sz, r_block_sz)),
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::tile<cuda_teams_y>(ctx, TILE_DIM, row_Range, [&] (RAJA::TypedRangeSegment
↪<int> const &row_tile) {

        RAJA::tile<cuda_teams_x>(ctx, TILE_DIM, col_Range, [&]␣
↪(RAJA::TypedRangeSegment<int> const &col_tile) {
```

(continues on next page)

(continued from previous page)

```
          RAJA::loop<cuda_threads_y>(ctx, row_tile, [&] (int row) {
            RAJA::loop<cuda_threads_x>(ctx, col_tile, [&] (int col) {

              Atview(col, row) = Aview(row, col);

            });
          });

        });
      });

  });
```

A notable difference between the CPU and GPU `RAJA::launch` implementations is the definition of the compute grid. For the CPU version, the argument list is empty for the `RAJA::LaunchParams` constructor. For the CUDA GPU implementation, we define a 'Team' of one two-dimensional thread-block with 16 x 16 = 256 threads.

Tiled Matrix Transpose with Local Array

This section extends the discussion in *Tiled Matrix Transpose* by adding *local array* objects which are used to store data for each tile in CPU stack-allocated arrays or GPU thread local and shared memory to be used within kernels.

There are exercise files `RAJA/exercises/kernel-matrix-transpose-local-array.cpp` and `RAJA/exercises/launch-matrix-transpose-local-array.cpp` for you to work through if you wish to get some practice with RAJA. The files `RAJA/exercises/kernel-matrix-transpose-local-array._solutioncpp` and `RAJA/exercises/launch-matrix-transpose-local-array_solution.cpp` contain complete working code for the examples. You can use the solution files to check your work and for guidance if you get stuck. To build the exercises execute `make (kernel/launch)-matrix-transpose-local-array` and `make (kernel/launch)-matrix-transpose-local-array_solution` from the build directory.

Key RAJA features shown in this example are:

- `RAJA::kernel_param` method and execution policy usage with multiple lambda expressions
- `RAJA::statement::Tile` type for loop tiling
- `RAJA::statement::ForICount` type for generating local tile indices
- `RAJA::LocalArray` type for thread-local tile memory arrays
- `RAJA::launch` kernel execution interface
- `RAJA::expt::tile` type for loop tiling
- `RAJA::expt::loop_icount` method to generate local tile indices for Launch
- `RAJA_TEAM_SHARED` macro for thread-local tile memory arrays

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, which 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 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. 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.

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

constexpr int TILE_DIM = 16;

constexpr int outer_Dimc = (N_c - 1) / TILE_DIM + 1;
constexpr 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-style 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
```

(continues on next page)

(continued from previous page)

```
          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` Variants

The `RAJA::kernel` interface provides mechanisms to tile loops and use *local arrays* in kernels so that algorithm patterns like the C-style kernel above can be implemented with RAJA. When using `RAJA::kernel`, a `RAJA::LocalArray` type specifies an object whose memory is created inside a kernel using a 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: the array data type, the 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::kernel` implementations of the matrix transpose operation.

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>
```

(continues on next page)

(continued from previous page)

```
          >
        >,

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


        >
      >
    >
  >;

RAJA::kernel_param<SEQ_EXEC_POL_I>(
  RAJA::make_tuple(RAJA::TypedRangeSegment<int>(0, N_c),
                   RAJA::TypedRangeSegment<int>(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);
  }

);
```

In the execution policy, the `RAJA::statement::Tile` types define tiling of the outer 'row' (iteration space tuple index '1') and 'col' (iteration space tuple index '0') loops, as well as tile sizes (`RAJA::tile_fixed` types) and loop execution policies. Next, the `RAJA::statement::InitLocalMem` type allocates the local tile 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 that are used in the two lambda expressions that comprise the kernel body. Finally, we have two sets of nested inner loops for reading the input matrix entries into the local tile array and writing them out to the output matrix transpose. The inner bodies of each of these loop nests are identified by lambda expression invocation statements `RAJA::statement::Lambda<0>` for the first lambda passed as an argument to the `RAJA::kernel_param` method and `RAJA::statement::Lambda<1>` for the second lambda argument.

Note that the loops within tiles use `RAJA::statement::ForICount` types rather than `RAJA::statement::For` types that we saw in the tiled matrix transpose example in *Tiled Matrix Transpose*. The `RAJA::statement::ForICount` type generates local tile indices that are passed to lambda loop body expressions to index into the local tile memory array. 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. 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. 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, which is similar to `RAJA::kernel` but takes additional arguments needed to execute the operations involving local tile indices and the local memory array. The first argument

is a tuple of iteration spaces that define the iteration ranges for the levels in the loop nest. Again, the first integer parameters given to the `RAJA::statement::Tile` and `RAJA::statement::ForICount` types identify the tuple entry to which they apply. The second argument:

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

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: `RAJA::kernel_param` accepts a parameter tuple argument after the iteration space tuple, which enables the parameters to be used in multiple lambda expressions in a kernel.

In the kernel, both lambda expressions take the same five arguments. The first two are the matrix global column and row indices associated with the iteration space tuple. The next three arguments correspond to the parameter tuple entries. The first two of these are the local tile indices used to access entries in the `RAJA::LocalArray` object memory. The last argument is a reference to the `RAJA::LocalArray` object itself.

The next `RAJA::kernel_param` variant we present works the same as the one above. It is different from the previous version since we include additional template parameters in the `RAJA::statement::Lambda` types to indicate which arguments each lambda expression 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::TypedRangeSegment<int>(0, N_c),
                     RAJA::TypedRangeSegment<int>(0, N_r)),

    RAJA::make_tuple(Tile_Array),

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

(continues on next page)

(continued from previous page)

```
        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 arguments; 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 in the previous example are not needed.

Note: In this example, we need all five arguments in each lambda expression so the lambda expression argument lists are the same. Another use case for the template parameter argument specification described here is to be able to pass only the arguments used in a lambda expression. In particular when we use multiple lambda expressions to represent a kernel, each lambda can have a different argument lists from the others.

`RAJA::expt::launch` Variants

The `RAJA::expt::launch` interface provides mechanisms to tile loops and use *local arrays* in kernels to support algorithm patterns like the C-style kernel above. When, using `RAJA::expt::launch`, the `RAJA_TEAM_SHARED` macro is used to create a GPU shared memory array or a CPU stack memory array inside a kernel.

`RAJA::expt::launch` support methods for tiling over an iteration space using `RAJA::expt::tile` and `RAJA::expt::loop_icount` methods to tile loops and generate global iteration indices and local tile offsets. Moreover, lambda expressions for these methods 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.

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

```
  using loop_pol_1 = RAJA::LoopPolicy<RAJA::loop_exec>;
  using launch_policy_1 = RAJA::LaunchPolicy<RAJA::seq_launch_t>;

  RAJA::launch<launch_policy_1>(
    RAJA::LaunchParams(), //LaunchParams may be empty when only running on the cpu
    [=] RAJA_HOST_DEVICE (RAJA::LaunchContext ctx) {

      RAJA::tile<loop_pol_1>(ctx, TILE_DIM, RAJA::TypedRangeSegment<int>(0, N_r), [&]␣
↪(RAJA::TypedRangeSegment<int> const &row_tile) {

        RAJA::tile<loop_pol_1>(ctx, TILE_DIM, RAJA::TypedRangeSegment<int>(0, N_c), [&
↪] (RAJA::TypedRangeSegment<int> const &col_tile) {

          RAJA_TEAM_SHARED double Tile_Array[TILE_DIM][TILE_DIM];
```

(continues on next page)

(continued from previous page)

```
        RAJA::loop_icount<loop_pol_1>(ctx, row_tile, [&] (int row, int ty) {
          RAJA::loop_icount<loop_pol_1>(ctx, col_tile, [&] (int col, int tx) {

            Tile_Array[ty][tx] = Aview(row, col);

          });
        });

        RAJA::loop_icount<loop_pol_1>(ctx, col_tile, [&] (int col, int tx) {
          RAJA::loop_icount<loop_pol_1>(ctx, row_tile, [&] (int row, int ty) {

            Atview(col, row) = Tile_Array[ty][tx];

          });
        });

      });
    });

  });
```

Here, the `RAJA::expt::tile` method is used to create tilings of the outer 'row' and 'col' iteration spaces. The `RAJA::expt::tile` method takes an additional argument specifying the tile size for the corresponding loop. To traverse the tile, we use the `RAJA::expt::loop_icount` method, which is similar to the `RAJA::ForICount` statement used in a `RAJA::kernel` execution policy as shown above. A `RAJA::expt::loop_icount` method call will generate local tile index associated with the outer global index. The local tile index is necessary as we use it to read and write entries from/to global memory to `RAJA_TEAM_SHARED` memory array.

Other RAJA Features and Usage Examples

Workgroup Constructs: Halo Exchange

The example code discussed in this section can be found in the file `RAJA/examples/tut_halo-exchange.cpp`. The file contains complete working code for multiple OpenMP, CUDA, and HIP RAJA variants. Here, we describe a subset of these variants.

Key RAJA features shown in this example:

- `RAJA::WorkPool` workgroup construct
- `RAJA::WorkGroup` workgroup construct
- `RAJA::WorkSite` workgroup construct
- `RAJA::TypedRangeSegment` 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 for distributed memory applications. This technique may not provide a performance gain on a CPU system, but it can significantly speedup halo exchange on a GPU system compared to running many individual packing/unpacking kernels, for example.

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 to the example code. These parameters determine the size of the grid, the width of the halo, the number of grid variables to pack/unpack, and the number of cycles; (iterations to run).

```
  //
  // 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 variable data arrays (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:

```
  //
  // Generate index lists for packing and unpacking
  //
  std::vector<int*> pack_index_lists(num_neighbors, nullptr);
  std::vector<int > pack_index_list_lengths(num_neighbors, 0);
  create_pack_lists(pack_index_lists, pack_index_list_lengths, halo_width, grid_dims);

  std::vector<int*> unpack_index_lists(num_neighbors, nullptr);
  std::vector<int > unpack_index_list_lengths(num_neighbors, 0);
  create_unpack_lists(unpack_index_lists, unpack_index_list_lengths, halo_width, grid_
↪dims);
```

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

Although the example code does not use MPI and multiple domains (one per MPI rank, for example), as would be the case in a real distributed memory parallel application, the data copy operations represent the spirit of how data communication would be done.

Packing and Unpacking (Basic Loop Execution)

A sequential non-RAJA example of data packing and unpacking would look like:

```
      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:

```
      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];

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

          buffer += len;
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

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];

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

      buffer += len;
    }
  }
```

For parallel multithreading 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>;
```

Note that we can use an asynchronous execution policy because there are no data races due to the intermediate buffer usage.

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,
                             RAJA::indirect_function_call_dispatch >;

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 messages are sent has been moved down after the call to run the operations enqueued 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) {

      double* var = vars[v];

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

      buffer += len;
    }
```

(continues on next page)

(continued from previous page)

```
    }

    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 multithreading 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,
                               RAJA::indirect_function_call_dispatch >;

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

  using workgroup = RAJA::WorkGroup< workgroup_policy,
                                     int,
                                     RAJA::xargs<>,
```

(continues on next page)

(continued from previous page)

```
                                      memory_manager_allocator<char> >;

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

The main differences between these types and the ones defined for the sequential case above are the `forall_policy` and the `workgroup_policy`, which use OpenMP execution policy types.

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,
                                RAJA::indirect_function_call_dispatch >;

   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 main differences between these types and the ones defined for the sequential and OpenMP cases above are the `forall_policy` and the `workgroup_policy`, which use different template parameters, and the `workpool`, `workgroup`, and `worksite` types which use 'pinned' memory allocation.

The packing is the same as the previous workgroup packing examples with the exception of added synchronization after calling the workgroup run method and before sending the messages. In the example code, there is a CUDA version that uses forall to launch `num_neighbors * num_vars` CUDA kernels and performs `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_policy` that reduces the number of CUDA kernel launches to one. It also allows us to need to synchronize only 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 with 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.

Matrix Multiplication: RAJA::kernel

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

raw CUDA and HIP versions of the kernel for comparison.

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
- 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 might look 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 matrix:

```
  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::TypedRangeSegment` objects to define the matrix row and column and dot product iteration spaces:

```
  RAJA::TypedRangeSegment<int> row_range(0, N);
  RAJA::TypedRangeSegment<int> col_range(0, N);
  RAJA::TypedRangeSegment<int> dot_range(0, N);
```

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 each 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 type of RAJA::kernel usage 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 shown 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 inner loop.

The integer arguments to the RAJA::statement::For types are needed to enable the desired kernel execution pattern and potential transformations, without changing the kernel code. Since the kernel policy is a single unified construct, it can be used to parallelize the nested loop iterations together, which we show next.

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 run sequentially:

```
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 *Basic RAJA::kernel Mechanics and Nested Loop Ordering*, we provide a more detailed discussion of the mechanics of loop nest ordering. 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,
        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) {
```

(continues on next page)

(continued from previous page)

```
      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). In general, the tuple is a set of *parameters* that can be used in the lambda expressions comprising the kernel. Here, the parameter tuple holds a single scalar variable for the dot product of each row-column pair.

The remaining arguments include a sequence of lambda expressions representing different parts of the kernel 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. This is the result of using the `RAJA::Params` and `RAJA::Segs` template parameter types in the `RAJA::statement::Lambda` types for lambdas '0' and '2'. Specifically, `RAJA::statement::Lambda<0, RAJA::Params<0>>` indicates that lambda '0' will take only the scalar parameter as an argument, and `RAJA::statement::Lambda<2, RAJA::Segs<0, 1>, RAJA::Params<0>>` indicates that lambda '2' will take index values for the column and row ranges and the scalar parameter as arguments, in that order. Since lambda '1' takes all arguments, we do not specify them.

Alternatively, the statement to invoke lambda '1' could be augmented to specify the arguments it takes:

```
// 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,
        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;
  },
```

(continues on next page)

(continued from previous page)

```
   // 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;
   }

 );
```

The result is the same.

By using `RAJA::statement::Lambda` parameters in this way, the code potentially indicates more clearly which arguments are used. Of course, this makes the execution policy more verbose, but that is typically hidden away in a header file, so it need not make the code harder to read.

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 may inspect the example code in the file noted above to see that this is indeed the case.

Next, we show how to collapse nested loops in an OpenMP parallel region using a `RAJA::statement::Collapse` type in the execution policy. This allows one to parallelize multiple levels in a loop nest using OpenMP directives. 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 += ...
       >,
       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. Here, '1, 0' indicates that the column loop is the inner loop and the row loop is the outer loop. 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 = ...
          >
        >
      >
    >;
```

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 += ...
```

(continues on next page)

(continued from previous page)

```
                >,
                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 execution 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 *Tiled Matrix Transpose with Local Array*, we discuss loop tiling in more detail.

6.2 Doxygen

6.3 RAJA Developer Guide

The RAJA Developer Guide documents software development processes followed by the RAJA project. The main goal of the guide is to ensure all project contributors understand the key elements of the processes so that they are consistently applied.

6.3.1 Contributing to RAJA

RAJA is a collaborative open source software project and we encourage contributions from anyone who wants to add features or improve its capabilities. This section describes the following:

- GitHub project access

- How to develop a RAJA *pull request* (PR) contribution.
- Requirements that must be met for a PR to be merged.

We assume contributors are familiar with Git, which we use for source code version control, and GitHub, which is where our project is hosted.

Important:

- Before a PR can be merged into RAJA, all test checks must pass and the PR must be approved by at least one member of the core RAJA team.
- Each RAJA contribution (feature, bugfix, etc.) must include adequate tests, documentation, and code examples. The *adequacy* of PR content, in this respect, is determined by PR reviewers applying their professional judgment considering the perspective of RAJA users and developers.

GitHub Project Access

RAJA maintains three levels of project access on it GitHub project:

- **Core team members.** Individuals on the core RAJA team are frequent RAJA contributors and participate regularly in project meetings, discussions, and other project activities. They are members of the LLNL GitHub organization and the `RAJA-core` GitHub team. Their project privileges include the ability to create branches in the repository, push code changes to the RAJA repo, make PRs, and merge them when they are approved and all checks have passed.
- **Regular contributors.** Individuals, who are not on the core RAJA team, but are members of the LLNL GitHub organization and are involved in some aspects of RAJA development are considered regular contributors. They are members of the `RAJA-contrib` GitHub team. Their project privileges include the ability to create branches in the repository, push code changes to the RAJA repo, and make PRs. However, they may not merge PRs and must coordinate with the core team to have their work included in the develop branch. This is mainly due to the way GitHub structures its project access levels.
- **Everyone else.** Anyone with a GitHub account is welcome to contribute to RAJA. Individuals outside of the two groups described above can make PRs in the RAJA project, but must do so from a branch on a *fork* of the RAJA repo. This is described below.

Pull Request Process

The following figure shows the basic elements of the RAJA PR contribution workflow. Some details vary depending on RAJA GitHub project access level of the contributor. The process involves four main steps:

1. A RAJA contributor makes a PR on the RAJA GitHub project to merge a branch on which she has developed a contribution into another RAJA branch, typically, the develop branch.
2. When a PR is created, GitHub triggers Azure CI testing checks and possibly Gitlab CI checks if the branch is part of the RAJA GItHub repo. Running and pass/fail status is reported back to GitHub where it can be viewed and monitored.
3. Meanwhile, RAJA team members and other contributors review the PR, suggesting changes and/or approving when they think it is ready to merge.
4. When all checks pass and the PR is approved, the PR may be merged.

This PR process should be familiar to nearly everyone who contributes to s software project. If you would like more information about pull requests, GitHub has a good PR guide on PR basics.

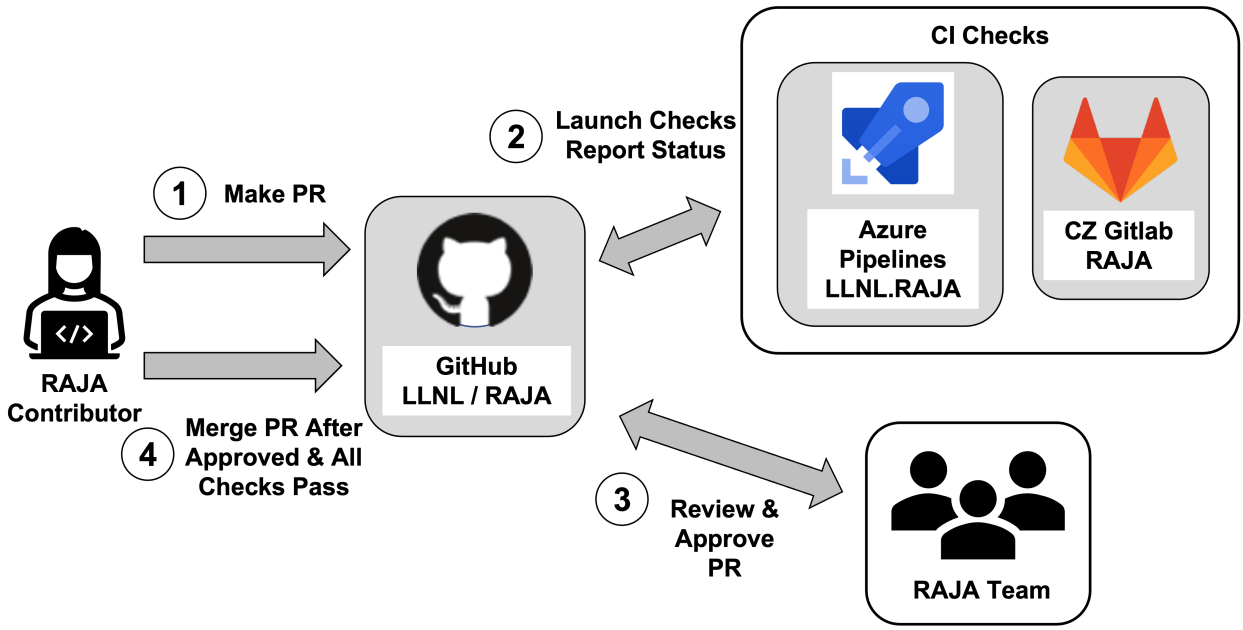

Fig. 8: The four main steps in the RAJA pull request (PR) process, which are common practices for many software projects.

Important: When you create a RAJA PR, you should enter a description of its contents in the *PR template* form the team maintains for this purpose. A good PR summary includes a descriptive title of the the bug you fixed or the feature you have added. Other relevant details that will assist others in reviewing your contribution should also be included.

Forking RAJA

As noted earlier, if you are not a member of the core RAJA development team, or a recognized RAJA contributor, then you do not have permission to create a branch in the RAJA GitHub repository. This choice is due to policies enforced by the LLNL organization on GitHub (in which the RAJA project resides) and the Livermore Computing (LC) organization (in which we run our Gitlab CI testing). 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 make changes on your local copy and push them your fork on GitHub. When you are ready to have your RAJA contribution reviewed ad added to the RAJA project, you may create a pull request in the RAJA project.

Note: A contributor who is not a member of the core RAJA development team, or a recognized RAJA contributor, cannot create a branch in the RAJA GitHub repo. However, anyone can create a fork of the RAJA project and create a pull request based on the fork in the RAJA project.

Developing A RAJA Contribution

New features, bugfixes, and other changes are developed on a **feature branch.** Each such branch should be based on the most current 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 a local, up-to-date copy of the `develop` branch by running the following commands:

```
$ git checkout develop
$ git pull origin develop
$ git submodule update --init --recursive
```

Then, in your local copy, you will be on the current version of develop branch with all RAJA submodules synchronized with that.

Feature and Bugfix Contributions

Assuming you are on an up-to-date develop branch in your local copy of RAJA, the first step toward developing a RAJA contribution is to create a new branch on which to do your development and push it to the remote origin of your local copy. For example:

```
$ git checkout -b <username>/feature/<name-of-feature>
$ git push <remote> <branch-name>
```

where `<username>/feature/<name-of-feature>` is the name of your feature branch. Or,

```
$ git checkout -b <username>/bugfix/<issue-fixed>
$ git push <remote> <branch-name>
```

where `<username>/bugfix/<issue-fixed>` is the name of your bugfix branch.

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, and examples and documentation, as needed. If you are creating new functionality, please add documentation to the appropriate section of the RAJA Documentation. The source files for the RAJA documentation are maintained in the `RAJA/docs` directory of the source repository. Consider adding example code(s) that illustrate usage of the new features you develop to help users and other developers understand your addition. These should be placed in the `RAJA/examples` directory and referenced in the RAJA User Guide as appropriate.

After your work is complete, you've tested it, and developed appropriate documentation, you can push your local branch to GitHub and create a PR in the RAJA project to merge your work into the RAJA develop branch. It will be reviewed by members of the RAJA team, who will provide comments, suggestions, etc.

As we stated earlier, not all required *Continuous Integration (CI) Testing* checks can be run on a PR made from a branch in a fork of RAJA. When the RAJA team has agreed to accept your work, it will be pulled into the RAJA GitHub repo (see *Accepting A Pull Request From A Forked Repository*). Then, it will run through all required testing and receive final reviews and approvals. When it is approved and all CI test checks pass, your contribution will be merged into the RAJA repository, most likely the develop branch.

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, **username/feature/<name-of-feature>** for a new feature, **username/bugfix/<issue-fixed>** for a bugfix, etc.

Accepting A Pull Request From A Forked Repository

Due to LLNL security policies, some RAJA pull requests will not be able to be run through all RAJA CI tools. The Livermore Computing (LC) Center Gitlab systems restrict which GitHub PRs may automatically run through its CI test pipelines. For example, a PR made from branch on a forked repository will not trigger Gitlab CI checks. Gitlab CI on LC platforms will be run only on PRs that are made from branches in the GitHub RAJA repository. See *Continuous Integration (CI) Testing* for more information about RAJA PR testing.

Note: The following process for accepting PR contributions from a fork of the RAJA repo must be executed by a member of the RAJA team:

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, which appears at the top of your PR. Then, run a 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 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, meaning that changes to them can 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 *RAJA Release Version Naming* for a description of the version labeling 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 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

A *feature* branch is created from another branch (usually develop) and is 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 almost never interact directly with the main branch. One exception is when a bug fix is needed in the main branch to tag a patch release.

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.

6.3.3 RAJA Build Configurations

To meet user needs, RAJA is built and tested with a wide range of compilers for all of its supported back-ends. Automated continuous integration (CI) testing employed by the project is described in *Continuous Integration (CI) Testing*. During day-to-day development, the project currently maintains two ways to build and test 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 a variety of platforms, such as Livermore Computing (LC) systems, MacOS, and Linux environments.
- **Generated host-config files.** The RAJA repository includes a mechanism to generate *host-config* files (i.e., CMake cache files) using Spack.

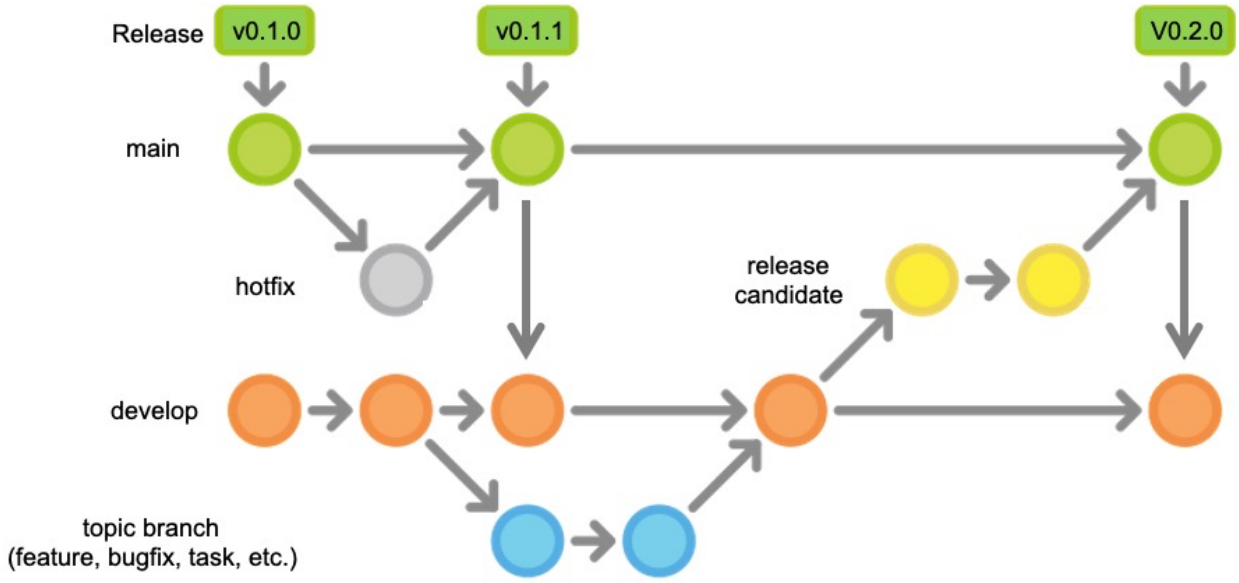

Fig. 9: This figure shows typical interactions between key branches in the Gitflow workflow. Here, development is shown following the v0.1.0 release. While development was ongoing, a bug was found and a fix was needed to be made available to users. A *hotfix* branch was made off of main and merged back 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.

The configurations specify compiler versions and options, build targets (Release, Debug, etc.), RAJA features to enable (OpenMP, CUDA, HIP, etc.), and paths to required tool chains, such as CUDA, ROCm, etc. They are described briefly in the following sections.

RAJA Build Scripts

Build scripts mentioned above live in the RAJA/scripts directory. Each script is executed from the top-level RAJA directory. The scripts for CPU-only platforms require an argument that indicates 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, the target compute architecture, and 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 build directory named for the configuration 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, and depending on options passed to CMake RAJA 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
```

Spack-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 and maintained by various projects. The main uberenv script is used to drive Spack to generate a *host-config* file (i.e., a CMake *cache* file) that contains all the information required to define a RAJA build environment. The generated 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.

Additional documentation for this process is available in the RADIUSS Uberenv Guide.

Generating a RAJA host-config file

This section describes the host-config file generation process for RAJA.

Platform configurations

Compiler configurations for Livermore computer platforms are contained in sub-directories of the `RAJA/scripts/radiuss-spack-configs` submodule directory:

```
$ ls -c1 ./scripts/radiuss-spack-configs
toss_4_x86_64_ib_cray
toss_4_x86_64_ib
toss_3_x86_64_ib
blueos_3_ppc64le_ib
darwin
config.yaml
blueos_3_ppc64le_ib_p9
...
```

To see available configurations, please see the contents of the `compilers.yaml` and `packages.yaml` files in each sub-directory.

Generating a host-config file

The `uberenv.py` python script can be run from the top-level RAJA directory to generate a host-config file for a desired configuration. For example,

```
$ python3 ./scripts/uberenv/uberenv.py --spec="%gcc@8.1.0"
$ python3 ./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
```

This process is also used by our Gitlab CI testing effort. See *Continuous Integration (CI) Testing* 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 a host-config file to CMake using the '-C' option. Then, run 'make' to build RAJA. To ensure the build was successful, you may want to run the RAJA tests. For example,

```
$ mkdir <build dirname> && cd <build dirname>
$ cmake -C <path_to>/<host-config>.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

You may also run the RAJA tests with the command

```
$ make test
```

as an alternative to the 'ctest' command used above.

It is also possible to use the configuration with the RAJA Gitlab CI script outside of the Gitlab environment:

```
$ 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 in the `compilers.yaml` file in the `RAJA/scripts/radiuss-spack-configs/darwin/` directory, and a commented section to illustrate how to add *CMake* as an external package in the `packages.yaml` in the same directory. You may also install CMake with Homebrew, for example, and follow the process outlined above after it is installed.

Reproducing Docker Builds Locally

RAJA uses Docker container images that it shares with other LLNL GitHub projects for Azure CI testing (see *Azure Pipelines CI* for more information). We use Azure Pipelines for Linux, Windows, and MacOS builds.

You can reproduce these builds locally for testing with the following steps if you have Docker installed.

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 a local directory in the image. For example

```
& docker -v pwd:/opt/RAJA
```

will mount your current local directory as `/opt/RAJA` in the image.

6.3.4 Continuous Integration (CI) Testing

Important:

- All CI test checks must pass before a pull request can be merged.
- The status (pass/fail and run) for all checks can be viewed by clicking the appropriate link in the **checks** section of a GitHub pull request.

The CI tools used by the RAJA project, and which integrate with GitHub are:

- **Azure Pipelines** runs builds and tests for Linux, Windows, and MacOS environments using recent versions of various compilers. While we do GPU builds for CUDA, HIP, and SYCL on Azure, RAJA tests are only run for CPU-only pipelines. See the RAJA Azure DevOps project to learn more about our testing there.
- **Gitlab** instances in the Livermore Computing (LC) Center runs builds and tests in LC resource and compiler environments important to many RAJA user applications. Execution of RAJA CI pipelines on LC Gitlab resources has restrictions described below. If you have access to LC resources, you can access additional information about LC GitLab CI

The tools automatically run RAJA builds and tests when a PR is created and when changes are pushed to a PR branch.

The following sections describe basic elements of the operation of the CI tools.

Gitlab CI

The Gitlab CI instance used by the RAJA project lives in the Livermore Computing (LC) Collaboration Zone (CZ). It runs builds and tests in LC resource and compiler environments important to RAJA user applications at LLNL.

Constraints

Running Gitlab CI on Livermore Computing (LC) resources is constrained by LC security policies. The policies require that all members of a GitHub project be members of the LLNL GitHub organization and have two-factor authentication enabled on their GitHub accounts to automatically mirror a GitHub repo and trigger Gitlab CI functionality from GitHub. For compliant LLNL GitHub projects, auto-mirroring of the GitHub repo on LC Gitlab is done when changes are pushed to PRs for branches in the RAJA repo, but not for PRs for a branch on a fork of the repo. An alternative procedure we use to handle this is described in *Contributing to RAJA*. If you have access to LC resources, you can learn more about LC Gitlab mirroring.

Gitlab CI (LC) Testing Workflow

The figure below shows the high-level steps in the RAJA Gitlab CI testing process. The main steps, which we will discuss in more detail later, are:

1. A *mirror* of the RAJA GitHub repo in the RAJA Gitlab project is updated whenever the RAJA `develop` or `main` branches are changed as well as when any PR branch in the RAJA GitHub project is changed.
2. Gitlab launches CI test pipelines. While running, the execution and pass/fail status may be viewed and monitored in the Gitlab CI GUI.
3. For each resource and compiler combination, Spack is used to generate a build configuration in the form of a CMake cache file, or *host-config* file.
4. A host-config file is passed to CMake, which configures a RAJA build space. Then, RAJA and its tests are compiled.

5. Next, the RAJA tests are run.
6. When a test pipeline completes, final results are reported in Gitlab.

In the next section, we will describe the roles that specific files in the RAJA repo play in defining these steps.

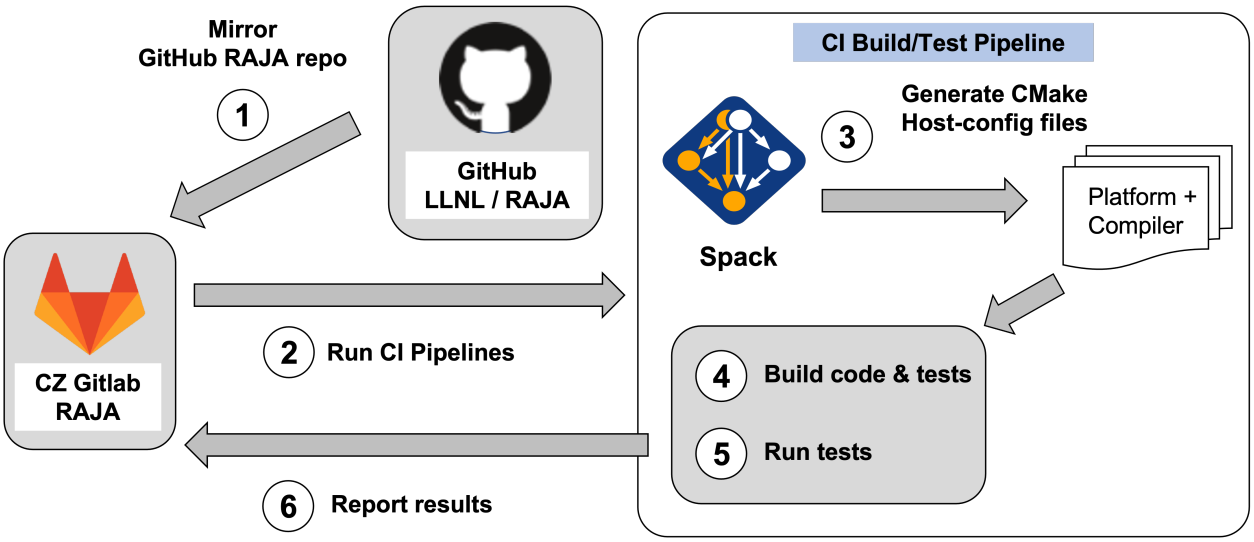

Fig. 10: The main steps in the RAJA Gitlab CI testing workflow are shown in the figure. This process is triggered when a developer makes a PR on the GitHub project or whenever changes are pushed to the source branch of a PR.

Gitlab CI (LC) Testing Files

The following figure shows directories and files in the RAJA repo that support LC Gitlab CI testing. Files with names in blue are specific to RAJA and are maintained by the RAJA team. Directories and files with names in red are in Git submodules, shared and maintained with other projects.

In the following sections, we discuss how these files are used in the steps in the RAJA Gitlab CI testing process summarized above.

Launching CI pipelines (step 2)

In **step 2** of the diagram above, Gitlab launches RAJA test pipelines. The RAJA/.gitlab-ci.yml file contains high-level testing information, such as stages (resource allocation, build-and-test, and resource deallocation) and locations of files that define which jobs will run in each pipeline. For example, these items appear in the file as:

```
stages:
  - r_allocate_resources
  - r_build_and_test
  - r_release_resources
  - l_build_and_test
  - c_build_and_test
  - multi_project
```

and:

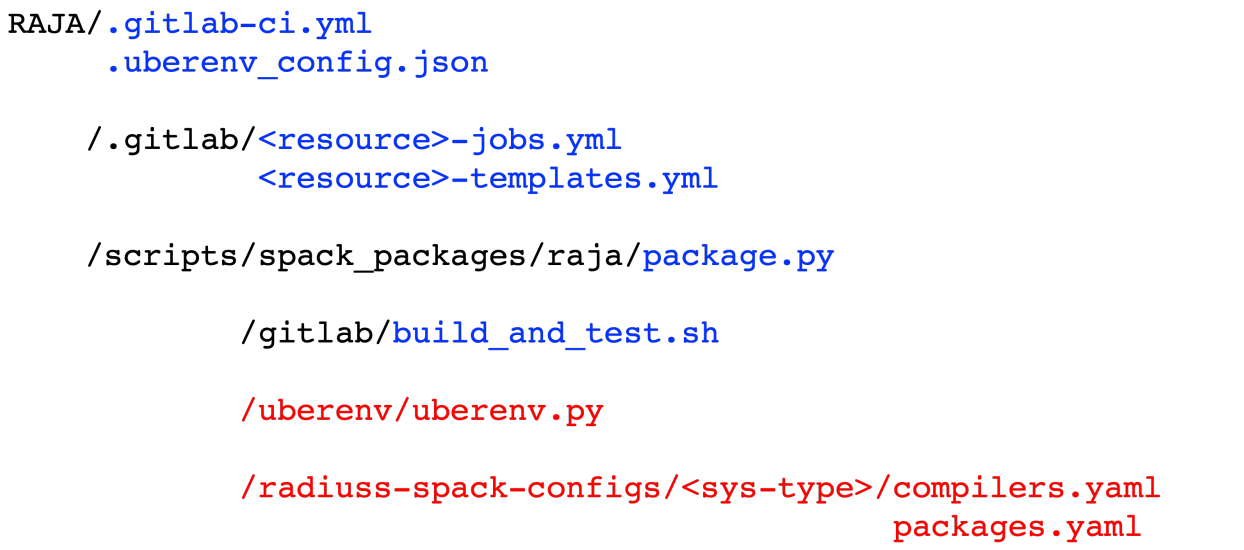

Fig. 11: The figure shows directories and files in the RAJA repo that support Gitlab CI testing. Files in blue are specific to RAJA and owned by the RAJA team. Red directories and files are part of Git submodules shared with other projects.

```
include:
  - local: .gitlab/ruby-templates.yml
  - local: .gitlab/ruby-jobs.yml
  - local: .gitlab/lassen-templates.yml
  - local: .gitlab/lassen-jobs.yml
  - local: .gitlab/corona-templates.yml
  - local: .gitlab/corona-jobs.yml
```

In the `stages` section above, prefixes '**r_**', '**l_**', and '**c_**' refer to resources in the LC on which tests are run. Specifically, the machines 'ruby', 'lassen', and 'corona', respectively. Jobs that will run in pipeline(s) on each resource are defined in the files listed in the `include` section above. Note that the stage labels above appear on each Gitlab CI run web page as the title of a column containing other information about what is run in that stage, such as build and test jobs.

The RAJA/.gitlab directory contains a *templates* and *jobs* file for each LC resource on which test pipelines will be run. The `<resource>-templates.yml` files contain information that is common across jobs that run on the corresponding resource, such as commands and scripts that are run for stages identified in the `RAJA/.gitlab-ci.yml` file. For example, the `RAJA/.gitlab/ruby-templates.yml` file contains a section:

```
allocate_resources (on ruby):
  variables:
    GIT_STRATEGY: none
  extends: .on_ruby
  stage: r_allocate_resources
  script:
    - salloc -N 1 -p pdebug -t 45 --no-shell --job-name=${ALLOC_NAME}
```

which contains the resource allocation command associated with the `r_allocate_resources` stage identifier on 'ruby'. Analogous stages are defined similarly in other `RAJA/.gitlab/<resource>-templates.yml` files.

The `<resource>-jobs.yml` files are described in the following sections.

Running a CI build/test pipeline (steps 3, 4, 5, 6)

The RAJA/scripts/gitlab/build_and_test.sh file defines the steps executed for each build and test run as well as information that will appear in the log output for each step. First, the script invokes the `RAJA/scripts/uberenv/uberenv.py` Python script located in the uberenv submodule:

```
...

python3 scripts/uberenv/uberenv.py --spec="${spec}" ${prefix_opt}

...
```

Project specific settings related to which Spack version to use, where Spack packages live, etc. are located in the RAJA/.uberenv_config.json file.

The uberenv python script invokes Spack to generate a CMake *host-config* file containing a RAJA build specification **(step 3)**. To generate a *host-config* file, Spack uses the RAJA Spack package, plus *Spack spec* information. The `RAJA/.gitlab/<resource>-jobs.yml` file defines a build specification (*Spack spec*) for each job that will be run on the corresponding resource. For example, in the `lassen-jobs.yml` file, you will see an entry such as:

```
gcc_8_3_1_cuda_10_1_168:
  variables:
    SPEC: "+cuda %gcc@8.3.1 cuda_arch=70 ^cuda@10.1.168"
  extends: .build_and_test_on_lassen
```

This defines the *Spack spec* for the test job in which CUDA device code will be built with the nvcc 10.1.168 compiler and non-device code will be compiled with the GNU 8.3.1 compiler. In the Gitlab CI GUI, this pipeline will be labeled `gcc_8_3_1_cuda_10_1_168`. Details for compilers, such as file system paths, target architecture, etc. are located in the `RAJA/scripts/radiuss-spack-configs/<sys-type>/compilers.yaml` file for the system type associated with the resource. Analogous information for packages like CUDA and ROCm (HIP) are located in the corresponding `RAJA/scripts/radiuss-spack-configs/<sys-type>/packages.yaml` file.

Note: Please see *Spack-Generated Host-Config Files* for more information about Spack-generated host-config files and how to use them for local debugging.

After the host-config file is generated, the `scripts/gitlab/build_and_test.sh` script creates a build space directory and runs CMake in it, passing the host-config (cache) file. Then, it builds the RAJA code and tests **(step 4)**:

```
...

build_dir="${build_root}/build_${hostconfig//.cmake/}"

...

date
echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
echo "~ Host-config: ${hostconfig_path}"
echo "~ Build Dir:   ${build_dir}"
echo "~ Project Dir: ${project_dir}"
echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
echo ""

echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
echo "~~~~~ Building RAJA"
echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
```

(continues on next page)

(continued from previous page)

```
rm -rf ${build_dir} 2>/dev/null
mkdir -p ${build_dir} && cd ${build_dir}

...

cmake \
  -C ${hostconfig_path} \
  ${project_dir}

cmake --build . -j ${core_counts[$truehostname]}
```

Next, it runs the tests **(step 5)**:

```
echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
echo "~~~~~ Testing RAJA"
echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"

...

cd ${build_dir}

...

ctest --output-on-failure -T test 2>&1 | tee tests_output.txt
```

Lastly, the script packages the test results into a JUnit XML file that Gitlab uses for reporting the results in its GUI **(step 6)**:

```
echo "Copying Testing xml reports for export"
tree Testing
xsltproc -o junit.xml ${project_dir}/blt/tests/ctest-to-junit.xsl Testing/*/Test.xml
mv junit.xml ${project_dir}/junit.xml
```

The commands shown here intermingle with other commands that emit messages, timing information for various operations, etc. which appear in a log file that can be viewed in the Gitlab GUI.

Azure Pipelines CI

The Azure Pipelines tool builds and tests for Linux, Windows, and MacOS environments. While we do builds for CUDA, HIP, and SYCL RAJA back-ends in the Azure Linux environment, RAJA tests are only run for CPU-only pipelines.

Azure Pipelines Testing Workflow

The Azure Pipelines testing workflow for RAJA is much simpler than the Gitlab testing process described above.

The test jobs we run for each OS environment are specified in the RAJA/azure-pipelines.yml file. This file defines the job steps, commands, compilers, etc. for each OS environment in the associated `- job:` section. A summary of the configurations we build are:

- **Windows.** The `- job: Windows` Windows section contains information for the Windows test builds. For example, we build and test RAJA as a static and shared library. This is indicated in the Windows `strategy` section:

```
strategy:
  matrix:
    shared:
      ...
    static:
      ...
```

We use the Windows/compiler image provided by the Azure application indicated the `pool` section; for example:

```
pool:
  vmImage: 'windows-2019'
```

MacOS. The `- job: Mac` section contains information for Mac test builds. For example, we build RAJA using the the MacOS/compiler image provided by the Azure application indicated in the `pool` section; for example:

```
pool:
  vmImage: 'macOS-latest'
```

Linux. The `- job: Docker` section contains information for Linux test builds. We build and test RAJA using Docker container images generated with recent versions of various compilers. The RAJA project shares these images with other open-source LLNL RADIUSS projects and they are maintained in the RES-ops Docker project on GitHub. The builds we do at any point in time are located in the `strategy` block:

```
strategy:
  matrix:
    gccX:
      docker_target: ...
    ...
    clangY:
      docker_target: ...
    ...
    nvccZ:
      docker_target: ...

    ...
```

The Linux OS the docker images are run on is indicated in the `pool` section; for example:

```
pool:
  vmImage: 'ubuntu-latest'
```

Docker Builds

For each Linux/Docker pipeline, the base container images, CMake, build, and test commands are located in RAJA/Dockerfile.

The base container images are built and maintained through the RSE-Ops RADIUSS project. A table of the most up to date containers can be found here. These images are rebuilt regularly ensuring that we have the most up to date builds of each container / compiler.

Note: Please see *Reproducing Docker Builds Locally* for more information about reproducing Docker builds locally for debugging purposes.

6.3.5 Continuous Integration (CI) Testing Maintenance Tasks

In *Continuous Integration (CI) Testing*, we described RAJA CI workflows. This section describes common CI testing maintenance tasks for RAJA and how to perform them.

Gitlab CI Tasks

The tasks in this section apply to GitLab CI running on Livermore Computing (LC) resources.

Changing Build Specs

The builds for each LC platform on which we run Gitlab CI pipelines are defined in `<resource>-jobs.yml` files in the RAJA/.gitlab directory. The key items that change when a new build is added are:

- the unique **label** that identifies the build on a web page for a Gitlab CI pipeline, and
- the build **Spack spec**, which identifies the compiler and version, compiler flags, etc.

For example, an entry for a build using a clang compiler with CUDA is:

```
ibm_clang_10_0_1_cuda_10_1_168:
  variables:
    SPEC: "+cuda cuda_arch=70 %clang@ibm.10.0.1 ^cuda@10.1.168"
  extends: .build_and_test_on_lassen
```

To update, change the corresponding spec item, such as clang compiler or version, or cuda version. Then, update the label accordingly.

It is important to note that the build spec information must reside in the `compilers.yaml` and/or `packages.yaml` file for the system type in the radiuss-spack-configs submodule. If the desired information is not there, try updating the submodule to a newer version. If the information is still not available, create a branch in the RADIUSS Spack Configs repo, add the needed spec info, and create a pull request.

Important: Build spec information used in RAJA Gitlab CI pipelines must exist in the `compilers.yaml` file and/or `packages.yaml` file for the appropriate system type in the RADIUSS Spack Configs repo.

Changing Build/Run Parameters

The commands executed to acquire resources on each system/system-type on which we run Gitlab CI are defined in the RAJA/.gitlab-ci.yml file. The default execution time for each test pipeline is also defined in the file using the variable `DEFAULT_TIME`. These commands and settings can remain as is for the most part.

However, sometimes a particular pipeline will take longer to build and run than the default allotted time. In this case, the default time can be adjusted in the build spec information in the associated `<resource>-jobs.yml` file discussed in the previous section. For example:

```
xl_16_1_1_7_cuda:
  variables:
    SPEC: "+cuda %xl@16.1.1.7 cuda_arch=70 ^cuda@10.1.168 ^cmake@3.14.5"
    DEFAULT_TIME: 60
  allow_failure: true
  extends: .build_and_test_on_lassen
```

This example explicitly sets the build and test allocation time to 60 minutes: `DEFAULT_TIME: 60`. Note that it also allows the pipeline to fail: `allow_failure: true`. We do this in some cases where certain tests are known to fail regularly. This allows the overall check status to report as passing, even though the test pipeline annotated this way may fail.

Adding Test Pipelines

Adding a test pipeline involves adding a new entry in the `RAJA/.gitlab-ci.yml` file.

Important: Build spec information used in RAJA Gitlab CI pipelines must exist in the `compilers.yaml` file and/or `packages.yaml` file for the appropriate system type in the RADIUSS Spack Configs repo.

Azure CI Tasks

The tasks in this section apply to RAJA Azure Pipelines CI.

Changing Builds/Container Images

The builds we run in Azure are defined in the RAJA/azure-pipelines.yml file.

Linux/Docker

To update or add a new compiler / job to Azure CI we need to edit both `azure-pipelines.yml` and `Dockerfile`.

If we want to add a new Azure pipeline to build with `compilerX`, then in `azure-pipelines.yml` we can add the job like so:

```
-job: Docker
  ...
  strategy:
    matrix:
      ...
      compilerX:
        docker_target: compilerX
```

Here, `compilerX:` defines the name of a job in Azure. `docker_target: compilerX` defines a variable `docker_target`, which is used to determine what part of the `Dockerfile` to run.

In the `Dockerfile` we will want to add our section that defines the commands for the `compilerX` job.:

```
FROM ghcr.io/rse-ops/compilerX-ubuntu-20.04:compilerX-XXX AS compilerX
ENV GTEST_COLOR=1
COPY . /home/raja/workspace
WORKDIR /home/raja/workspace/build
RUN cmake -DCMAKE_CXX_COMPILER=compilerX ... && \
    make -j 6 &&\
    ctest -T test --output-on-failure
```

Each of our docker builds is built up on a base image maintained by RSE-Ops, a table of available base containers can be found here. We are also able to add target names to each build with `AS ...`. This target name correlates to the `docker_target: ...` defined in `azure-pipelines.yml`.

The base containers are shared across multiple projects and are regularly rebuilt. If bugs are fixed in the base containers the changes will be automatically propagated to all projects using them in their Docker builds.

Check here for a list of all currently available RSE-Ops containers. Please see the RSE-Ops Containers Project on Github to get new containers built that aren't yet available.

Windows / MacOs

We run our Windows / MacOS builds directly on the Azure virtual machine instances. In order to update the Windows / MacOS instance we can change the `pool` under `-job: Windows` or `-job: Mac`:

```
-job: Windows
  ...
  pool:
    vmImage: 'windows-2019'
  ...
-job: Mac
  ...
  pool:
    vmImage: 'macOS-latest'
```

Changing Build/Run Parameters

Linux/Docker

We can edit the build and run configurations of each docker build, in the `RUN` command. Such as adding CMake options or changing the parallel build value of `make -j N` for adjusting throughput.

Each base image is built using spack. For the most part the container environments are set up to run our CMake and build commands out of the box. However, there are a few exceptions where we need to `spack load` specific modules into the path.

- **Clang** requires us to load LLVM for OpenMP runtime libraries.:

  ```
  . /opt/spack/share/spack/setup-env.sh && spack load llvm
  ```

  CUDA for the cuda runtime.:

  ```
  . /opt/spack/share/spack/setup-env.sh && spack load cuda
  ```

  HIP for the hip runtime and llvm-amdgpu runtime libraries.:

  ```
  . /opt/spack/share/spack/setup-env.sh && spack load hip llvm-amdgpu
  ```

  SYCL requires us to run setupvars.sh:

  ```
  source /opt/view/setvars.sh
  ```

Windows / MacOS

Windows and MacOS build / run parameters can be configured directly in `azure-pipelines.yml`. CMake options can be configured with `CMAKE_EXTRA_FLAGS` for each job. The `-j` value can also be edited directly in the Azure `script` definitions for each job.

The commands executed to configure, build, and test RAJA for each pipeline in Azure are located in the RAJA/Dockerfile file. Each pipeline section begins with a line that ends with `AS ...` where the ellipses in the name of a build-test pipeline. The name label matches an entry in the Docker test matrix in the `RAJA/azure-pipelines.yml` file mentioned above.

RAJA Performance Suite CI Tasks

The RAJA Performance Suite project CI testing processes, directory/file structure, and dependencies are nearly identical to that for RAJA, which is described in *Continuous Integration (CI) Testing*. Specifically,

- The RAJA Performance Suite Gitlab CI process is driven by the RAJAPerf/.gitlab-ci.yml file.
- The `<resource>-jobs.yml` and `<resource>-templates.yml` files reside in the RAJAPerf/.gitlab directory.
- The `build_and_test.sh` script resides in the RAJAPerf/scripts/gitlab directory.
- The RAJAPerf/Dockerfile drives the Azure testing pipelines.

The main difference is that for Gitlab CI, is that the Performance Suite uses the RAJA submodules for `uberenv` and `radiuss-spack-configs` located in the RAJA submodule to avoid redundant submodules. This is reflected in the RAJAPerf/.uberenv_config.json file which point at the relevant RAJA submodule locations.

Apart from this minor difference, all CI maintenance and development tasks for the RAJA Performance Suite follow the guidance in *Continuous Integration (CI) Testing Maintenance Tasks*.

6.3.6 RAJA Tests

As noted in *Continuous Integration (CI) Testing*, all RAJA test checks must pass before any PR contribution will be merged. Additionally, we recommend that contributors include new tests in their code contributions when adding new features and bug fixes.

Note: If RAJA team members think adequate testing is not included in a PR branch, they will ask for additional testing to be added during the review process.

Test Organization

The goals of the RAJA test organization are to:

- Make it easy to see what is tested and where tests live. We want developers and users to be able to find tests easily and know where to put new tests when they add them.
- Parameterize tests as much as reasonable to ensure that features work with all supported RAJA back-ends and we are testing them consistently. We want the source files for each test case to allow testing of each RAJA back-end. Specifically, tests for each back-end are generated by instantiating the same source routines with different type information.
- Have test source code generated for compilation by CMake when the code is configured. This significantly reduces code redundancy and enables our test parameterization goals.

All RAJA tests reside in the RAJA/test directory. The test directory structure looks like this:

```
RAJA/test/functional/forall
                     kernel
                     scan
                     ...
          include/...
          integration/...
          unit/algorithm
               atomic
               index
               ...
```

RAJA tests are partitioned into three main categories:

- **Unit tests** exercise basic interfaces and features of individual RAJA classes and methods in standalone fashion; i.e., integrated with other parts of RAJA as minimally as is reasonable. RAJA unit tests reside in sub-directories of the RAJA/test/unit directory.
- **Functional tests** integrate multiple RAJA features in common ways to test how RAJA is used in practice. RAJA functional tests reside in sub-directories of the RAJA/test/functional directory.
- **Integration tests** exercise features that integrate RAJA with other libraries, such as Kokkos performance tools as plug-ins. RAJA integration tests reside in sub-directories of the RAJA/test/integration directory.

The RAJA/test/include directory contains header files that define types and other items that are commonly used in various tests.

Important: Please follow the existing sub-directory structure and code implementation patterns for RAJA tests when adding or modifying tests.

Anatomy Of A Test Case

This section discusses in some detail the structure of files for a single RAJA test case and how the work together. In particular, we describe the set of basic tests that exercise `RAJA::forall` execution with various RAJA segment types.

Note: The implementation pattern described in the following sections is similarly used by all other RAJA tests.

Since these tests integrate multiple RAJA features, it is considered a *functional* test. The files for this test are located in the RAJA/test/functional/forall/segment directory. The contents of the directory are:

```
$ ls -c1 -R ./test/functional/forall/segment
./test/functional/forall/segment:
tests
test-forall-segment.cpp.in
CMakeLists.txt

./test/functional/forall/segment/tests:
test-forall-RangeStrideSegment.hpp
test-forall-RangeSegment.hpp
test-forall-ListSegment.hpp
```

Next, we describe these and their relationships.

Test Source File

The test-forall-segment.cpp.in file is the parameterized test source file. It contains header file include statements:

```
//
// test/include headers
//
#include "RAJA_test-base.hpp"
#include "RAJA_test-camp.hpp"
#include "RAJA_test-index-types.hpp"

#include "RAJA_test-forall-data.hpp"
#include "RAJA_test-forall-execpol.hpp"

//
// Header for tests in ./tests directory
//
// Note: CMake adds ./tests as an include dir for these tests.
//
#include "test-forall-@SEGTYPE@.hpp"
```

The first set of header files live in the `RAJA/test/include` directory mentioned earlier. The headers are centrally located since their contents are shared with other test files. The last include statement pulls in the header file containing the parameterized tests for the corresponding RAJA segment type.

Next, a `camp::cartesian_product` type is defined to assemble sets of types used in the parameterized tests:

```
//
// Cartesian product of types used in parameterized tests
//
using @BACKEND@ForallSegmentTypes =
  Test< camp::cartesian_product<StrongIdxTypeList,
                                @BACKEND@ResourceList,
                                @BACKEND@ForallExecPols>>::Types;
```

The first template argument defining the `camp::cartesian_product object` type refers to a list of segment index types defined in the RAJA_test-index-types.hpp header file. The second argument refers to a list of RAJA/camp resource types appropriate for the RAJA execution back-end defined in the RAJA_test-camp.hpp header file (see *Test Header files* for where this is used). The third argument refers to a list of RAJA execution policy types defined in the RAJA_test-forall-execpol.hpp header file. This results in the generation of a combinatorial collection of typed tests being run. Each test is defined by a unique tuple of types, described in *Test Header files*.

Lastly, the parameterized set of tests is instantiated:

```
//
// Instantiate parameterized test
//
INSTANTIATE_TYPED_TEST_SUITE_P(@BACKEND@,
                               Forall@SEGTYPE@Test,
                               @BACKEND@ForallSegmentTypes);
```

`INSTANTIATE_TYPED_TEST_SUITE_P` is a GoogleTest macro. The first argument is a label noting the RAJA back-end used for the generated tests. This can be used to filter the tests when they are manually run. The second argument is a label identifying the test set, and the third argument matches the CMake generated name for the `camp::cartesian_product` type described above.

Important: The second argument passed to the `INSTANTIATE_TYPED_TEST_SUITE_P` macro must match the

name of the test suite class discussed in *Test Header files*.

CMakeLists.txt File

The concrete version of each of the items described above is generated by CMake when a RAJA build is configured. CMake fills in the segment type and back-end identifiers, `@SEGTYPE@` and `@BACKEND@`, respectively. These identifiers and the test file and executable generation process is defined in the CMakeLists.txt file in the test directory. If you look in the file, you will see nested loops over RAJA back-ends and segment types which process the test source file `test-forall-segment.cpp.in` multiple times to create a uniquely named source file for each back-end/segment type combination in the RAJA build space. Each source file will be compiled into a similarly named, unique test executable when the code is compiled.

Test Header files

Recall the line in the test source file:

```
#include "test-forall-@SEGTYPE@.hpp"
```

This identifies the header file containing the actual test code used to generate the tests. The test header files are located in the RAJA/test/functional/forall/segment/tests directory. The main elements of each test header file are described next. We use the test-forall-RangeSegment.hpp file to illustrate the essential test implementation elements.

The file contains the following important items:

- test implementation method
- typed test suite class
- typed test invocation
- type test suite registration

The test implementation is contained in a parameterized template method:

```
template <typename INDEX_TYPE, typename WORKING_RES, typename EXEC_POLICY>
void ForallRangeSegmentTestImpl(INDEX_TYPE first, INDEX_TYPE last)
{
   ...
}
```

Here, the template parameters identify the index type of the RAJA segment `INDEX_TYPE`, the resource type for allocating test memory in the proper execution environment `WORKING_RES`, and the execution policy `EXEC_POLICY` for the `RAJA::forall` method used to run the tests.

The test suite class plugs into the GoogleTest framework:

```
TYPED_TEST_SUITE_P(ForallRangeSegmentTest);
template <typename T>
class ForallRangeSegmentTest : public ::testing::Test
{
};
```

using the `TYPED_TEST_SUITE_P` GoogleTest macro.

Important: The name of the test class must be identical to the label passed to the GoogleTest `TYPED_TEST_SUITE_P` macro.

The specific tests that are run are defined by calls to the test implementation template method `ForallRangeSegmentTestImpl` described above:

```
TYPED_TEST_P(ForallRangeSegmentTest, RangeSegmentForall)
{
  using INDEX_TYPE  = typename camp::at<TypeParam, camp::num<0>>::type;
  using WORKING_RES = typename camp::at<TypeParam, camp::num<1>>::type;
  using EXEC_POLICY = typename camp::at<TypeParam, camp::num<2>>::type;

  // test zero-length range segment
  ForallRangeSegmentTestImpl<INDEX_TYPE, WORKING_RES, EXEC_POLICY>(INDEX_TYPE(3),␣
↪INDEX_TYPE(3));

  ForallRangeSegmentTestImpl<INDEX_TYPE, WORKING_RES, EXEC_POLICY>(INDEX_TYPE(0),␣
↪INDEX_TYPE(27));
  ForallRangeSegmentTestImpl<INDEX_TYPE, WORKING_RES, EXEC_POLICY>(INDEX_TYPE(1),␣
↪INDEX_TYPE(2047));
  ForallRangeSegmentTestImpl<INDEX_TYPE, WORKING_RES, EXEC_POLICY>(INDEX_TYPE(1),␣
↪INDEX_TYPE(32000));

  runNegativeTests<INDEX_TYPE, WORKING_RES, EXEC_POLICY>();
}
```

Here, `TYPED_TEST_P` is a GoogleTest macro defining the method for executing the tests. Note that the first three lines in the method extract the template parameter types from the `camp::tuple` produced by the `camp::cartesian_product` described earlier in *Test Source File*. If you look in the file, you will see an example of how we use C++ SFINAE to exclude running tests with negative index values for index types that are unsigned.

Important:

- The label passed as the first argument to the GoogleTest `TYPED_TEST_P` macro must match the name of the test suite class. The second argument is discussed below.
- It is critical to use the same type ordering when extracting the types that was used when the `camp::cartesian_product` type was defined in the test source file, described in *Test Source File*.

Lastly, the test suite is registered with GoogleTest using the `REGISTER_TYPED_TEST_SUITE_P` macro:

```
REGISTER_TYPED_TEST_SUITE_P(ForallRangeSegmentTest,
                            RangeSegmentForall);
```

Important:

- The label passed as the first argument to the GoogleTest `REGISTER_TYPED_TEST_SUITE_P` macro must match the name of the test suite class.
- The label passed as the second argument to the GoogleTest `REGISTER_TYPED_TEST_SUITE_P` macro must match the label passed as the second argument to the `TYPED_TEST_P` macro.

6.3.7 RAJA Release Process

RAJA is considered part of the **RAJA Portability Suite** set of projects. Currently, the Suite includes Umpire, CHAI, and camp, in addition to RAJA.

Important: Releases for the Suite are coordinated, meaning that when a non-patch release is done for one, a new version release is done for all Suite projects.

The RAJA release process includes 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.
5. On GitHub, make a new release with a tag for the release. Following our convention, the tag label should have the format `YYYY.mm.pp`. See *RAJA Release Version Naming* 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 may cause issues for users as a result.

   Important: For consistency, please follow a similar release 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 be included in the develop branch and future work on that branch.

After a RAJA release is done, there a other tasks that typically need to be performed to update content in other projects. These task are described in *Post-release Activities*.

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 that may negatively impact RAJA users. A hotfix branch will address the issue be merged into both develop and main branches.

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. Update the release notes and RAJA patch version number as described in *Release Candidate Branch*.
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 `YYYY.mm.pp`, where only the **patch** portion of the release tag should differ from the last release. 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 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.

Post-release Activities

After a RAJA release is complete, other tasks are performed to update content in other repositories, typically. These tasks include:

- Update the RAJAProxies project to the newly RAJA Portability Suite projects. This typically consists of updating the submodules to the new RAJA Portability Suite project versions, making sure the proxy-apps build and run correctly. When this is done, tag a release for proxy-app project.
- Update the RAJA Spack package in the Spack repository. This requires some knowledge of Spack and attention to details and Spack conventions. Please see *Spack Package Update* for details.

Spack Package Update

Describe how to update the RAJA Spack package. . . .

6.3.8 RAJA Release Version Naming

Prior to the RAJA release in March 2022, the RAJA project used the *Semantic Versioning* scheme for assigning release tag names. At the March 2022 release, we changed the release naming scheme to use `YYYY.mm.pp`, for year, month, and patch number. So, for example, the March 2022 release is labeled v2022.03.0.The main motivation for the release naming scheme is to do coordinated releases with the Umpire, CHAI, and camp projects, which are considered parts of the **RAJA Portability Suite**. In a coordinated release, all the projects will have the same release name. If a project requires a patch release between coordinated releases, it will indicate that by incrementing the patch number; for example, v2022.03.1.

The following sections describe the Semantic Versioning scheme for reference and posterity.

Semantic Versioning

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.

Semantic 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 Semantic 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

6.4.1 RAJA License