# An Overview of RAJA

LLNL internal RAJA tutorial

April 15, 2020

EXASCALE COMPUTING PROJECT

**Rich Hornung (hornung1@llnl.gov)**
**With contributions from the rest of the RAJA Team**

LLNL-PRES-781841

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

# Welcome

- Today, we will describe RAJA features and how it enables performance portability
- We will also discuss some topics that may be helpful when thinking about how to achieve portability in applications
- Our objective for today is that you learn enough to start using RAJA in your own code development

See the RAJA User Guide for more information (linked on RAJA GitHub project).

# During the presentation…

**Please don’t hesitate to ask questions at any time**

# We value your feedback…

- If you have comments, questions, or suggestions, please let us know
  - Send a message to our project email list: raja-dev@llnl.gov
- We appreciate specific, concrete feedback that helps us improve RAJA and how we present it to users

# RAJA and performance portability

- RAJA is a **library of C++ abstractions** that enable you to write **portable, single-source** kernels – run on different hardware by re-compiling
  - Multicore CPUs, Xeon Phi, NVIDIA GPUs, …
- RAJA **insulates application source code** from hardware and programming model-specific implementation details
  - OpenMP, CUDA, SIMD vectorization, …
- RAJA supports a variety of **parallel patterns** and **performance tuning** options
  - Simple and complex loop kernels
  - Reductions, scans, atomic operations, multi-dim data views for changing access patterns, …
  - Loop tiling, thread-local data, GPU shared memory, …

RAJA provides building blocks that extend the generally-accepted **"parallel for" idiom**.

# RAJA design goals target usability and developer productivity

- We want applications to maintain **single-source kernels** (as much as possible)
- In addition, we want RAJA to…
  - Be **easy to understand and use** for app developers (esp. those who are not CS experts)
  - Allow **incremental and selective adoption**
  - **Not force major disruption** to application source code
  - Promote flexible algorithm implementations via **clean encapsulation**
  - Make it **easy to parameterize execution** via type aliases
  - Enable **systematic performance tuning**

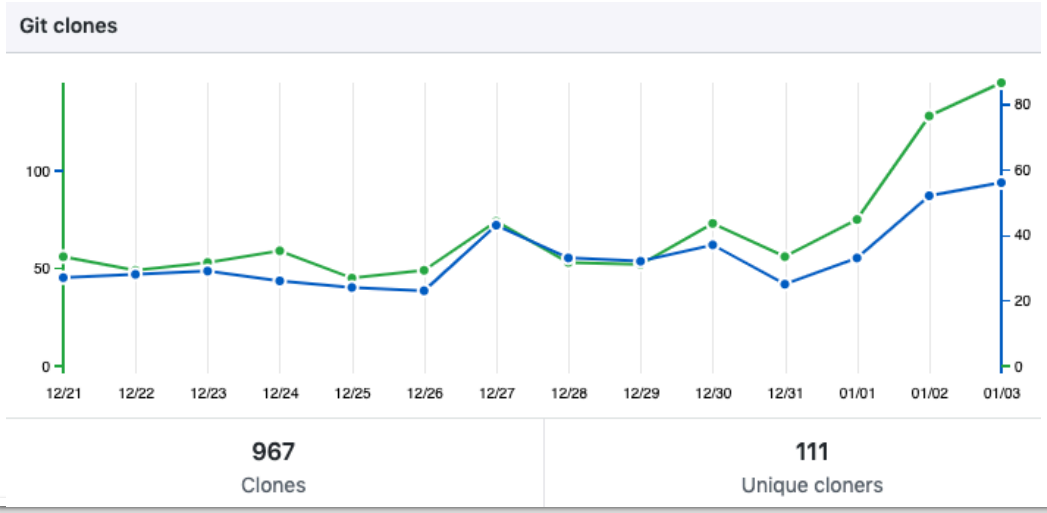
These goals have been affirmed by production LLNL application teams using RAJA.

# RAJA is an open source project (https://github.com/LLNL/RAJA)

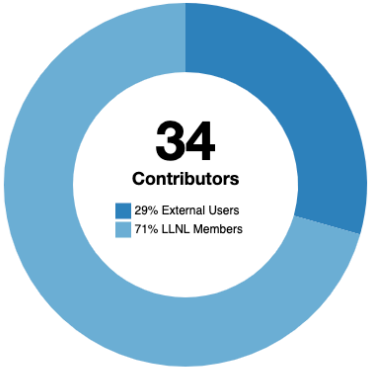
RAJA

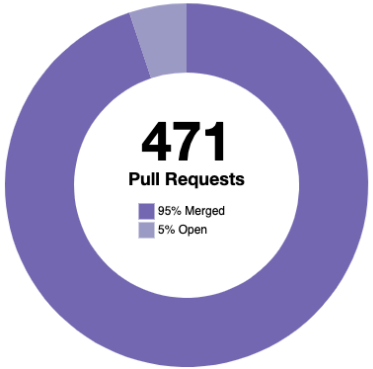
LLNL | C++ | BSD-3-Clause

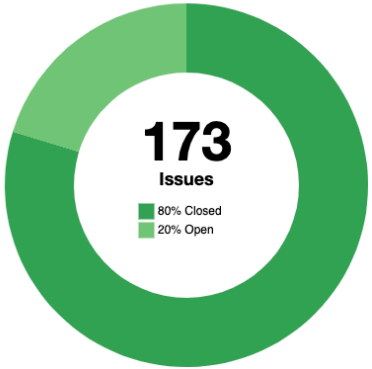
GitHub Page  Stargazers : 161  Forks : 47




GitHub activity
1/19 – 1/20

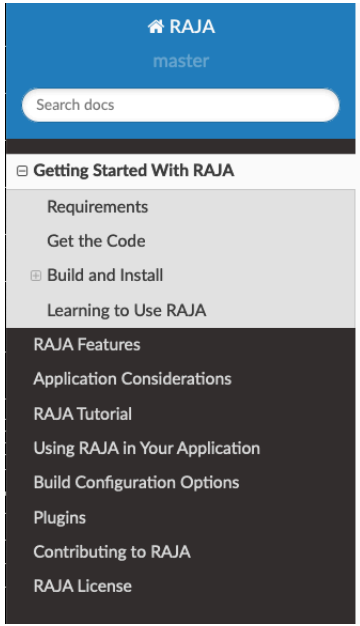


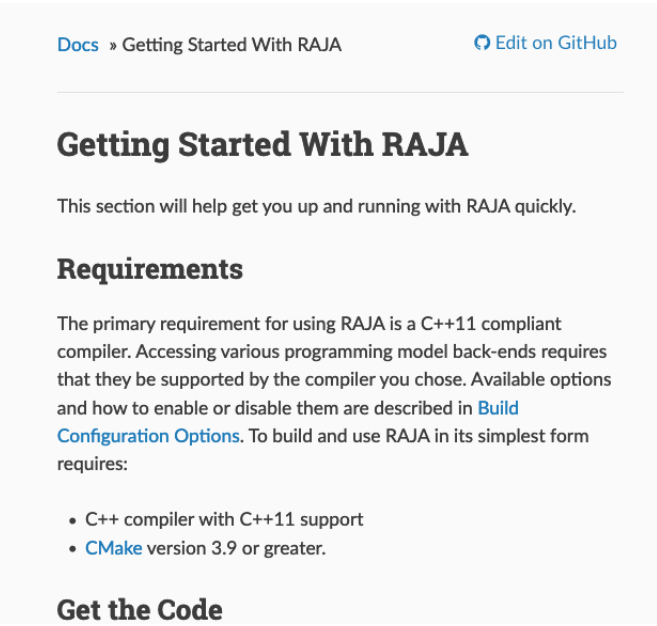



# We maintain other related open source projects…

- **RAJA User Guide:** getting started info, details about features and usage, etc. (readthedocs.org/projects/raja)
- **RAJA Project Template:** shows how to use RAJA in an application that uses CMake or Make (https://github.com/LLNL/RAJA-project-template)
- **RAJA Performance Suite:** loop kernels for assessing compilers and RAJA performance. Used by us, vendors, for DOE platform procurements, etc. (https://github.com/LLNL/RAJAPerf)
- **CHAI:** array abstraction library that automatically migrates data as needed based on RAJA execution contexts (https://github.com/LLNL/CHAI)




All of these are linked on the RAJA GitHub project page.

# We will cover a range of topics today

- RAJA usage considerations (C++ templates, lambdas, memory management, etc.)
- Generally useful information for reasoning about how to prepare your code to be portable to run on diverse architectures
- RAJA features:
  - Simple loops
  - Reductions
  - Iteration spaces
  - Atomic operations
  - Scan operations
  - Data layouts and views
  - Complex loop kernels, including some advanced topics

# Let's start simple…

## Simple loop execution

# Consider a typical C-style for-loop…

"daxpy" operation: y = a * x + y, where x, y are vectors of length N, a is a scalar

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

Note that all aspects of execution are explicit in the source code – execution (sequential), loop iteration order, data access pattern, etc.

# Converting a loop to RAJA mainly involves changing the loop header



C-style for-loop

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

"RAJA Transformation"

RAJA-style loop

```
RAJA::forall< EXEC_POL >( it_space, [=] (int i)
{
   y[i] = a * x[i] + y[i];
} );
```

# RAJA encapsulates loop execution details

C-style for-loop

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

Typically, definitions like these go in header files.

```
using EXEC_POL = ...;

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

RAJA-style loop

```
RAJA::forall< EXEC_POL >( it_space, [=] (int i)
{
   y[i] = a * x[i] + y[i];
} );
```

By changing the “execution policy” and “iteration space”, you change the way the loop runs.

# The loop header is different with RAJA, but the loop body is the same (in most cases)



C-style for-loop

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

RAJA-style loop

```
using EXEC_POL = ...;

RAJA::RangeSegment it_space(0, N);

RAJA::forall< EXEC_POL >( it_space, [=] (int i)
{
  y[i] = a * x[i] + y[i];
} );
```

Same loop body.

# RAJA loop execution has four core concepts

```
using EXEC_POLICY = ...;
RAJA::RangeSegment range(0, N);

RAJA::forall< EXEC_POLICY >( range, [=] (int i)
{
   // loop body...
} );
```

1. Loop **execution template** (e.g., ‘forall’)
2. Loop **execution policy type** (EXEC_POLICY)
3. Loop **iteration space** (e.g., ‘RangeSegment’)
4. Loop **body** (C++ lambda expression)

# RAJA loop execution core concepts

```
RAJA::forall< EXEC_POLICY > ( iteration_space,
   [=] (int i) {
      // loop body
   }
);
```

- RAJA::forall method runs loop based on:
  - **Execution policy type** (sequential, OpenMP, CUDA, etc.)

# RAJA loop execution core concepts

```
RAJA::forall< EXEC_POLICY > ( iteration_space,
  [=] (int i) {
     // loop body
  }
);
```

- RAJA::forall template runs loop based on:
  - Execution policy type (sequential, OpenMP, CUDA, etc.)
  - **Iteration space object** (stride-1 range, list of indices, etc.)

# These core concepts are common threads throughout our discussion



```
RAJA::forall< EXEC_POLICY > ( iteration_space,
  [=] (int i) {
     // loop body
  }
);
```

- RAJA::forall template runs loop based on:
  - Execution policy type (sequential, OpenMP, CUDA, etc.)
  - Iteration space object (contiguous range, list of indices, etc.)
- **Loop body is cast as a C++ lambda expression**
  - Lambda argument is the loop iteration variable

The programmer must ensure the loop body works with the execution policy; e.g., thread safe

# The execution policy determines the programming model back-end



```
RAJA::forall< EXEC_POLICY >( range, [=] (int i)
{
   x[i] = a * x[i] + y[i];
} );
```

```
RAJA::simd_exec
```

```
RAJA::omp_parallel_for_exec
```

```
RAJA::cuda_exec<BLOCK_SIZE, Async>
```

```
RAJA::omp_target_parallel_for_exec<MAX_THREADS_PER_TEAM>
```

```
RAJA::tbb_for_exec
```

A sample of RAJA loop execution policy types.

# RAJA provides a variety of execution policy types...

- Sequential (forces strictly sequential execution)
- “Loop” (lets compiler decide which optimizations to apply)
- SIMD (applies vectorization pragmas)
- OpenMP multithreading (CPU)
- TBB** (Intel Threading Building Blocks)
- CUDA (NVIDIA GPUs)
- OpenMP target** (offload to target device; e.g., GPU)
- HIP** (AMD GPUs)

**Some execution back-ends are **works-in-progress.**

**Note that basic RAJA usage is conceptually the same as a C-style for-loop. The syntax is different.**

# Before we continue, let's discuss some C++ features you need to understand to use RAJA effectively

# RAJA makes heavy use of C++ templates

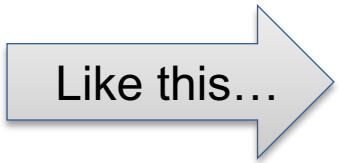
```
template <typename ExecPol,
          typename IdxType,
          typename LoopBody>
forall(IdxType&& idx, LoopBody&& body) {
   ...
}
```

- Templates represent *generic* code. The *compiler generates* an implementation for each set of template parameter types specified.
- Here, “ExecPol”, “IdxType”, “LoopBody” are C++ types you provide at compile-time

# RAJA makes heavy use of C++ templates

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

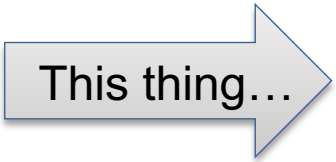
- You specify “ExecPol”, “IdxType”, “LoopBody” are types




```
forall< seq_exec >( RangeSegment(0, N), ...
   // loop body
);
```

- Note: “ExecPol” type is explicit, whereas “IdxType” and “LoopBody” types are deduced by the compiler

# You pass a loop body to RAJA as a C++ lambda expression (C++11)




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

- A lambda expression is a *closure* that stores a function with a data environment
- It is like a functor, but much easier to use

# C++ lambda expressions…

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

- A lambda expression has the following form

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

# Lambda expression concepts…

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

- A lambda expression has the following form

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

- The capture list specifies how variables (in enclosing scope) are pulled into the lambda data environment
  - By-value or by-reference ( **[=]** vs. **[&]** )? By-value required for GPU execution, RAJA reductions ….
  - **We recommend using capture by-value in all cases**, as shown above

# Lambda expression concepts…

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

- A lambda expression has the following form

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

- The capture list specifies how variables are pulled into the lambda data environment
  - We recommend using capture by-value in all cases
- The parameter list contains arguments passed to lambda function body

# Lambda expression concepts…

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

- A lambda expression has the following form

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

- The capture list specifies how variables are pulled into the lambda data environment
  - We recommend using capture by-value in all cases
- The parameter list arguments are passed to lambda function body
- A lambda passed to a CUDA kernel requires a *device annotation*:

[=] **__device__** (…) { … }

# Lambda expression concepts…

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

- A lambda expression has the following form

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

- The capture list specifies how variables (outer scope) are pulled into lambda data environment
  - We recommend using capture by-value in all cases
- The parameter list are arguments passed to lambda function body; e.g., `(int i)` is "loop variable"
- A lambda passed to a CUDA kernel requires a device annotation: [=] **__device__** (…) { … }

The RAJA User Guide has more information about C++ lambda expressions.

# Back to RAJA features…

# Reductions

# Reduction is a common and important parallel pattern

dot product: $dot = \sum_{i=0}^{N-1} a_i \, b_i$ , where a and b are vectors, dot is a scalar

C-style

```
double dot = 0.0;
for (int i = 0; i < N; ++i) {
  dot += a[i] * b[i];
}
```

# A RAJA reduction object hides the complexity of a parallel reduction operation

C-style

```
double dot = 0.0;
for (int i = 0; i < N; ++i) {
  dot += a[i] * b[i];
}
```

RAJA

```
RAJA::ReduceSum< REDUCE_POLICY, double> dot(0.0);

RAJA::forall< EXEC_POLICY >( range, [=] (int i) {
  dot += a[i] * b[i];
} );
```

# Elements of RAJA reductions…

```
RAJA::ReduceSum< REDUCE_POLICY, DTYPE > sum(init_val);

RAJA::forall< EXEC_POLICY >(... {
  sum += func(i);
});

DTYPE reduced_sum = sum.get();
```

- A **reduction type** requires:
  - A reduction policy
  - A reduction value type
  - An initial value

# Elements of RAJA reductions…

```
RAJA::ReduceSum< REDUCE_POLICY, DTYPE > sum(init_val);

RAJA::forall< EXEC_POLICY >(... {
  sum += func(i);
});

DTYPE reduced_sum = sum.get();
```

Note that you cannot access the reduced value inside a kernel.

- A reduction type requires:
  - A reduction policy
  - A reduction value type
  - An initial value
- **Updating reduction value is what you expect (+=, min, max)**

# Elements of RAJA reductions…

```
RAJA::ReduceSum< REDUCE_POLICY, DTYPE > sum(init_val);

RAJA::forall< EXEC_POLICY >(... {
  sum += func(i);
});

DTYPE reduced_sum = sum.get();
```

- A reduction type requires:
  - A reduction policy
  - A reduction value type
  - An initial value
- Updating reduction value is what you expect (+=, min, max)
- **After loop runs, get reduced value via 'get' method**

# The reduction policy must be compatible with the loop execution policy

```
RAJA::ReduceSum< REDUCE_POLICY, DTYPE > sum(init_val);

RAJA::forall< EXEC_POLICY >(... {
  sum += func(i);
});

DTYPE reduced_sum = sum.get();
```

An OpenMP execution policy requires an OpenMP reduction policy, similarly for CUDA, etc.

# RAJA provides reduction policies for all supported programming model back-ends

```
RAJA::ReduceSum< REDUCE_POLICY, int > sum(0);
```

```
RAJA::seq_reduce;
RAJA::omp_reduce;
RAJA::cuda_reduce;
RAJA::tbb_reduce;
RAJA::omp_target_reduce;
```

A sample of RAJA reduction policy types.

Note: OpenMP target and HIP reductions are works-in-progress.

# RAJA supports five common reductions types

```
RAJA::ReduceSum<    REDUCE_POLICY, DTYPE > r(in_val);

RAJA::ReduceMin<    REDUCE_POLICY, DTYPE > r(in_val);

RAJA::ReduceMax<    REDUCE_POLICY, DTYPE > r(in_val);

RAJA::ReduceMinLoc< REDUCE_POLICY, DTYPE > r(in_val,
                                             in_loc);

RAJA::ReduceMaxLoc< REDUCE_POLICY, DTYPE > r(in_val,
                                             in_loc);
```

Initial "loc" values

"Loc" reductions give a loop index where reduced value was found.

# Multiple RAJA reductions can be used in a kernel

```
RAJA::ReduceSum< REDUCE_POL, int > sum(0);
RAJA::ReduceMin< REDUCE_POL, int > min(MAX_VAL);
RAJA::ReduceMax< REDUCE_POL, int > max(MIN_VAL);
RAJA::ReduceMinLoc< REDUCE_POL, int > minloc(MAX_VAL, -1);
RAJA::ReduceMaxLoc< REDUCE_POL, int > maxloc(MIN_VAL, -1);

RAJA::forall< EXEC_POL >( RAJA::RangeSegment(0, N), [=](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);
} );
```

# Suppose we run the code on the previous slide with this array…

‘a’ is an int vector of length 12 initialized as:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a : | 1 | -1 | 1 | -1 | 1 | -5 | 5 | -5 | 1 | -1 | 1 | -1 |

- *What are the reduced values?*
  - *Sum?*
  - *Min?*
  - *Max?*
  - *Max-loc?*
  - *Min-loc?*

# Suppose we run the code on the previous slide with this array…

‘a’ is an int vector of length 12 initialized as:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a : | 1 | -1 | 1 | -1 | 1 | -5 | 5 | -5 | 1 | -1 | 1 | -1 |

- *What are the reduced values?*
  - Sum = -4
  - Min = -5
  - Max = 5
  - Max-loc = 6
  - Min-loc = 5 or 7

In general, the result of a parallel reduction is order-dependent.

# Iteration spaces : Segments and IndexSets

# A RAJA “Segment” defines a set of loop iterates

Iteration spaces

- A **Segment** is a set of loop indices to run for a kernel

**Contiguous range** [beg, end)

**Strided range** [beg, end, stride)

**List of indices** (indirection)

# Loop iteration spaces are defined by Segments

Iteration spaces

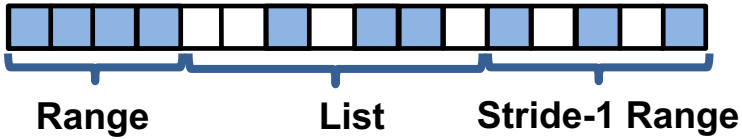
- A Segment is a set of loop indices to run for a kernel

**Contiguous range** [beg, end)

**Strided range** [beg, end, stride)

**List of indices** (indirection)

- An **Index Set** is a container of segments (of arbitrary types)




You can run all Segments in an IndexSet in one call to a RAJA loop execution template.

# A RangeSegment defines a contiguous sequence of indices (stride-1)

Iteration spaces

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

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

Runs loop indices: 0, 1, 2, … , N-1

# A RangeStrideSegment defines a strided sequence of indices

Iteration spaces

```
RAJA::RangeStrideSegment srange1( 0, N, 2 );

RAJA::forall< RAJA::seq_exec >( srange1 , [=] (int i)
{
   // ...
} );
```

Runs loop indices: 0, 2, 4, …

# Segments also allow negative indices and strides

Iteration spaces

```
RAJA::RangeStrideSegment srange2( N-1, -1, -1 );

RAJA::forall< RAJA::seq_exec >( srange2 , [=] (int i)
{
   // ...
} );
```

Runs loop in reverse: N-1, N-2, … , 1, 0

# Segments are templates on the index type



RangeSegment and RangeStrideSegment are **type aliases**

```
using RAJA::RangeSegment =
        RAJA::TypedRangeSegment<RAJA::Index_type>;

using RAJA::RangeStrideSegment =
        RAJA::TypedRangeStrideSegment<RAJA::Index_type>;
```

# Segments are templates on the index type



RangeSegment and RangeStrideSegment are **type aliases**

```
using RAJA::RangeSegment =
        RAJA::TypedRangeSegment<RAJA::Index_type>;

using RAJA::RangeStrideSegment =
        RAJA::TypedRangeStrideSegment<RAJA::Index_type>;
```

- RAJA::IndexType is a useful parametrization
  - It is an alias to std::ptrdiff_t
  - **Appropriate for most compiler optimizations**

Use the 'Typed' Segment types for other index value types.

# A ListSegment can define any set of indices

Iteration spaces

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

// array of indices
IdxType idx[ ] = {10, 11, 14, 20, 22};

// ListSegment object containing indices...
ListSegType idx_list( idx, 5 );
```

Think *“indirection array”.*

# A ListSegment can define any set of indices

Iteration spaces

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

// array of indices
IdxType idx[ ] = {10, 11, 14, 20, 22};

// ListSegment object containing indices...
ListSegType idx_list( idx, 5 );

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

Runs loop indices: 10, 11, 14, 20, 22

Note: indirection **does not** appear in loop body.

# A RAJA IndexSet is a container of Segments

Iteration spaces

```
using RangeSegType = RAJA::TypedRangeSegment<RAJA::Index_type>;
using ListSegType  = RAJA::TypedListSegment<RAJA::Index_type>;

RangeSegType range1(0, 8);

RAJA::Index_type idx[ ] = {10, 11, 14, 20, 22};
ListSegType list2( idx, 5 );

RangeSegType range3(24, 28);
```

# An IndexSet may contain different Segment types

Iteration spaces

```
using RangeSegType = RAJA::TypedRangeSegment<RAJA::Index_type>;
using ListSegType = RAJA::TypedListSegment<RAJA::Index_type>;

RangeSegType range1(0, 8);

RAJA::Index_type idx[ ] = {10, 11, 14, 20, 22};
ListSegType list2( idx, 5 );

RangeSegType range3(24, 28);

RAJA::TypedIndexSet< RangeSegType, ListSegType > iset;

iset.push_back( range1 );
iset.push_back( list2 );
iset.push_back( range3 );
```

Segment types must be specified at compile time

# IndexSets enable iteration space partitioning

Iteration spaces

```
using RangeSegType = RAJA::TypedRangeSegment<RAJA::Index_type>;
using ListSegType = RAJA::TypedListSegment<RAJA::Index_type>;

RangeSegType range1(0, 8);

RAJA::Index_type idx[ ] = {10, 11, 14, 20, 22};
ListSegType list2( idx, 5 );

RangeSegType range3(24, 28);

RAJA::TypedIndexSet< RangeSegType, ListSegType > iset;

iset.push_back( range1 );
iset.push_back( list2 );
iset.push_back( range3 );
```

**Iteration space is partitioned into 3 Segments**

| 0, …, 7 | , | 10, 11, 14, 20, 22 | , | 24, …, 27 |
|---|---|---|---|---|
| range1 | | list2 | | range3 |

# All segments in an IndexSet can be run in a single call to a RAJA execution template

Iteration spaces

```
using ISET_EXECPOL =
   RAJA::ExecPolicy< RAJA::omp_parallel_segit,
                     RAJA::seq_exec >;

RAJA::forall<ISET_EXECPOL>(iset, [=] (IdxType i) {
  // loop body
} );
```

Index sets require a **two-level execution policy**:

- Outer iteration over segments ( "…_segit")
- Inner segment execution

# Why does RAJA provide Index Sets?

Iteration spaces

- **Multiphysics codes use indirection arrays** (a lot!)
  - Indirection inhibits performance: more instructions + memory traffic, impedes optimizations
- **Range Segments are better for performance**
  - When large stride-1 ranges are embedded in an iteration space…
  - …you can expose these as SIMD-izable ranges “in place” to compilers (no gather/scatters)
- **Partitioning and reordering iterations gives flexibility and performance**
  - Avoid fine-grained synchronization (atomics or critical sections), which are **contention heavy**
  - Avoid extra arrays and gather/scatter operations, which require **extra memory traffic**
  - Prefer coarse-grained synchronization, which has much **lighter memory contention**

IndexSets give a lot of flexibility for kernel iteration patterns **without changing the way a kernel looks** in source code.

# Atomic operations

# RAJA provides portable atomic operations

```
// two pointers to 'int' memory locations
int* x = ...
int* y = ...

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

  RAJA::atomicAdd< ATOMIC_POLICY >(x, 1); // atomically add 1 to x
  RAJA::atomicSub< ATOMIC_POLICY >(y, 1); // atomically subtract
                                          // 1 from y

} );
```

An atomic operation updates a specific memory address (write or read-modify-write) such that only one thread or process at a time can write to it.

# RAJA OpenMP atomic approximation of pi

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

double* pi = new double[1]; *pi = 0.0;
double dx = 1.0 / N;

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

  double x = ( double(i) + 0.5 ) * dx;
  RAJA::atomicAdd< ATOMIC_POL >( pi, dx / (1.0 + x * x) );

} );
*pi *= 4.0;
```

$$\frac{\pi}{4} = \tan^{-1}(1) = \int_0^1 \frac{1}{1+x^2}\,dx$$

The atomic policy must be compatible with the loop execution policy (like reductions).

# The RAJA “builtin” atomic policy uses compiler built-in atomics



```
using EXEC_POL = RAJA::omp_parallel_for_exec;

int *sum = ...;

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

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

} );
```

# The RAJA "auto" atomic policy will pick the correct atomic implementation

```
using EXEC_POL = RAJA::omp_parallel_for_exec;

int *sum = ...;

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

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

} );
```

Some may prefer this option for simpler portability.

# RAJA also has an interface modeled after the C++20 standard std::atomic_ref feature

- "AtomicRef" supports:
  - Arbitrary memory locations
  - All RAJA atomic policies

**For example:**

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

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

Result: sum is 5 (= 2 + 1 + 1 + 1).

# RAJA provides a variety of atomic operations

- Arithmetic: add, sub
- Min, max
- Increment/decrement: inc, dec, including conditional comparisons with other values
- Bitwise-logical: and, or, xor
- Replace: exchange, compare-and-swap (CAS)
- C++ std::atomic_ref style interface (RAJA::AtomicRef)

The RAJA User Guide describes the full set of RAJA atomics.

# Scan operations

# Scan is an important building block for parallel algorithms

- It is a key primitive for developing parallel algorithms
  - Based on reduction tree and reverse reduction tree
  - Scan is an example of a computation that looks inherently serial, but for which there exist efficient parallel implementations

- Many useful applications:
  - Sorting (radix, quicksort)
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrence relations
  - Tree operations
  - Histograms
  - Parallel work assignment

Useful reference:
"Prefix Sums and Their Applications" by Guy E. Blelloch
(https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf)

# Prefix sum is the most common scan operation

```
int* in  = ...;  // input array of length N
int* out = ...;  // output array of length N
```

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

```
RAJA::exclusive_scan< EXEC_POL >(in, in + N, out);
```

**Example:**

**In :** 8 -1 2 9 10 3 4 1 6 7 (N=10)

**Out (inclusive) :** 8 7 9 18 28 31 35 36 42 49

**Out (exclusive) :** 0 8 7 9 18 28 31 35 36 42

Note: Exclusive scan shifts the result array one slot to the right. The first entry of an exclusive scan is the identity of the scan operator; here it is “+”.

The output array contains partial sums of input array.

# RAJA also provides “in-place” scan operations

```
int* arr  = ...;  // in/out array of length N
```

```
RAJA::inclusive_scan_inplace< EXEC_POL >(arr, arr + N);
```

```
RAJA::exclusive_scan_inplace< EXEC_POL >(arr, arr + N);
```

“In-place” scans return the result in the input array.

# RAJA provides different operators to use in scans

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

**In :** 8 -1 2 9 10 -3 4 1 6 7

**Out :** 2147483648 8 -1 -1 -1 -1 -3 -3 -3 -3

*What is the first value in the result of this scan?*

If no operator is given, “plus” is the default (prefix-sum).

# RAJA provides different operators to use in scans

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

**In :** 8 -1 2 9 10 -3 4 1 6 7

**Out :** 2147483648 8 -1 -1 -1 -1 -3 -3 -3 -3

*What is the first value in the result of this scan?*

It is the identity (I) of the minimum operator – min(I, value) = value

If no operator is given, “plus” is the default (prefix-sum).

# Views and Layouts

# Matrices and tensors are common in scientific computing

View/Layout

- Most naturally, they are thought of as multi-dimensional arrays, but for efficiency in C/C++ they are usually allocated as 1-d arrays

```
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) {
         C[col + N*row] += A[k + N*row] * B[col + N*k];
      }
      C[col + N*row] = dot;
   }
}
```

C-style matrix multiplication

- Here, we manually convert 2-d indices (row, col) to pointer offsets

# RAJA Views and Layouts make multi-dimensional indexing simpler and more flexible

View/Layout

- A RAJA View wraps a pointer to enable indexing that follows a prescribed Layout pattern

```
const int DIM = 2;
double* A = new double[ N * N ];

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

# RAJA Views and Layouts make multi-dimensional indexing simpler and more flexible

- A RAJA View wraps a pointer to enable indexing that follows a prescribed Layout pattern

```
const int DIM = 2;
double* A = new double[ N * N ];

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

- This leads to data indexing that is simpler, more intuitive, and less error-prone

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

# RAJA Views and Layouts support any number of dimensions

View/Layout

```
const int DIM = n + 1;
double* A = new double[ N0 * ... * Nn ];

View< double, Layout<DIM> > Aview(A, N0, ..., Nn);

// iterate over nth index and hold others fixed
for (int j = 0; j < Nn; ++j) {
  Aview(i0, i1, ..., j) = ...;
}

// iterate over jth index and hold others fixed
for (int j = 0; j < Nj; ++j) {
  Aview(i0, i1, ..., j, ..., iN) = ...;
}
```

Stride-1 data access

Data access stride is Nn * … * N(j+1)

A default RAJA layout Layout<DIM> uses ‘row-major’ ordering.
The right-most index is stride-1.

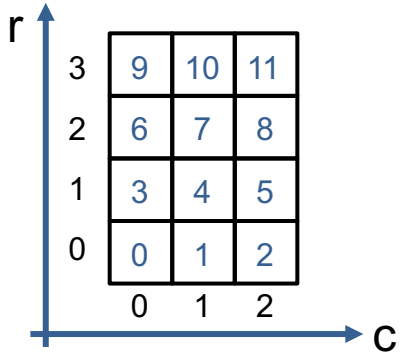
# Every Layout has a permutation

```
std::array<RAJA::idx_t, 2> perm {{0, 1}};  // default permutation

RAJA::Layout< 2 > perm_layout =
  RAJA::make_permuted_layout( {{4, 3}}, perm); // r, c extents

double* a = ...;
RAJA::View< double, RAJA::Layout<2, int> > Aview(A, perm_layout);

Aview(r, c) = ...;
```




"c" index has stride 1
(rightmost in permutation)

"r" index has stride 3
(extent of "c" index)

# And so on for higher dimensions…

```
std::array<RAJA::idx_t, 3> perm {{1, 2, 0}};

RAJA::Layout< 3 > perm_layout =
  RAJA::make_permuted_layout( {{5, 7, 11}}, perm);

RAJA::View< double, RAJA::Layout<3> > Bview(B, perm_layout);

// Equivalent to indexing as: B[i + j*5*11 + k*5]
Bview(i, j, k) = ...;
```

3-d layout with indices permuted:

- Index ‘0’ (i) has stride 1 and extent 5
- Index ‘2’ (k) has stride 5 and extent 11
- Index ‘1’ (j) has stride 55 (= 5 * 11) and extent 7

Permutations enable you to alter the access pattern to improve performance.

# An offset layout applies an offset to indices

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

RAJA::OffsetLayout<1> offlayout =
  RAJA::make_offset_layout<1>( {{-5}}, {{5}} );

RAJA::View< double, RAJA::OffsetLayout<1> > Cview(C,
                                                  offlayout);

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

A 1-d View with index offset and extent 11 [-5, 5].

-5 is subtracted from each loop index to access data.

Offset layouts are useful for index space subset operations such as halo regions.

# Important notes about RAJA Layout types

- There is no RAJA::PermutedLayout type (since every layout has a permutation):

```
RAJA::Layout< NDIMS > perm_layout =
  RAJA::make_permuted_layout( ... );
```

- RAJA::OffsetLayout is a distinct type (since it has a RAJA::Layout and offset data):

```
RAJA::OffsetLayout< NDIMS > offset_layout =
  RAJA::make_offset_layout( ... );

RAJA::OffsetLayout< NDIMS > perm_offset_layout =
  RAJA::make_permuted_offset_layout( ... );
```

# Offset layout quiz…

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

- *What index space does this layout represent?*

# Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

# Offset layout quiz…

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

- *What index space does this layout represent?*

  The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

# Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Index ‘1’ (right-most) is stride-1 (using default permutation).

# Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Index ‘1’ (right-most) is stride-1 (using default permutation).

- *What is the stride of index ‘0’?*

# Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Index ‘1’ (right-most) is stride-1 (default permutation).

- *What is the stride of index ‘0’?*

Index ‘0’ has stride 11 (since index 1 has extent 11, [-5, 5]).

# Let's try a permuted offset layout…

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

- *What index space does this layout represent?*

# Let's try a permuted offset layout…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

# Let's try a permuted offset layout…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

# Let's try a permuted offset layout…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to permutation).

# Let's try a permuted offset layout…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to permutation).

- *What is the stride of index '1'?*

# Let's try a permuted offset layout…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to permutation).

- *What is the stride of index '1'?*

Index '1' has stride 4 (since index '0' has extent 4, [-1, 2]).

# RAJA layouts have methods to convert between multi-dimensional indices and linear indices

View/Layout

```
RAJA::Layout<3> layout(5, 7, 11);
```

A 3-d layout with extents 5, 7, 11.

# RAJA layout methods convert between multi-dimensional indices and linear indices



```
RAJA::Layout<3> layout(5, 7, 11);
```

A 3-d layout with “(i, j, k)” extents 5, 7, 11.

```
// Convert i=2, j=3, k=1 to linear index
int lin = layout(2, 3, 1);
```

*What is the value of “lin”?*

# RAJA layout methods convert between multi-dimensional indices and linear indices



```
RAJA::Layout<3> layout(5, 7, 11);
```

A 3-d layout with “(i, j, k)” extents 5, 7, 11.

```
// Convert i=2, j=3, k=1 to linear index
int lin = layout(2, 3, 1);
```

*What is the value of “lin”?*

lin = 188 (= 1 + 3 * 11 + 2 * 11 * 7)

# RAJA layout methods convert between multi-dimensional indices and linear indices



```
RAJA::Layout<3> layout(5, 7, 11);
```

A 3-d layout with “(i, j, k)” extents 5, 7, 11.

```
// Convert linear index 191 to (i,j,k) index
layout.toIndices(191, i, j, k);
```

*What is the (i, j, k) index?*

# RAJA layout methods convert between multi-dimensional indices and linear indices



```
RAJA::Layout<3> layout(5, 7, 11);
```

A 3-d layout with “(i, j, k)” extents 5, 7, 11.

```
// Convert linear index 191 to (i,j,k) index
layout.toIndices(191, i, j, k);
```

*What is the (i, j, k) index?*

(i, j, k) = (2, 3, 4)

191 (= **4** + **3** * 11 + **2** * 11 * 7)

RAJA provides a compile configuration option to check at run time whether indices are in bounds. See the User Guide for details.

# Complex loops and (some) advanced RAJA features

# Nested loops

# Recall the matrix multiplication example…

C = A * B, where A, B, C are N x N matrices

C-style nested for-loops

```
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[k + N*row] * B[col + N*k];
    }
    C[col + N*row] = dot;

  }
}
```

# We could try to use nested ‘forall’ statements to make a RAJA implementation…

Nested loops

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

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

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

  } );
} );
```

Note: we use RAJA Views to simplify multi-dimensional indexing.

# …But this doesn't work well

- ***Each loop level is treated as an independent entity***
  - How to parallelize the row and column loops together?
- We can parallelize the outer row loop (OpenMP, CUDA, etc.)
  - But then, each thread executes all code in the inner two loops sequentially
- Parallelizing the inner column loop introduces unwanted synchronization
  - We launch a new parallel computation for each row
- Loop interchange and other transformations require changing the source code of the kernel (which breaks RAJA encapsulation)

Bottom line: don't use RAJA::forall for nested loops!!

# The RAJA::kernel API is designed for composing and transforming complex kernels

Nested loops

```
using namespace RAJA;
using KERNEL_POL = KernelPolicy<
                     statement::For<1, exec_policy_row,
                       statement::For<0, exec_policy_col,
                         statement::Lambda<0>
                       >
                     >
                   >;

RAJA::kernel<KERNEL_POL>( RAJA::make_tuple(col_range, row_range),
                          [=](int col, int row ) {
    double dot = 0.0;
    for (int k = 0; k < N; ++k) {
      dot += A(row, k) * B(k, col);
    }
    C(row, col) = dot;
  } );
```

Note: lambda expression for inner loop body is the same as C-style variant.

# The RAJA::kernel interface uses four basic concepts, analogous to RAJA::forall

Nested loops

1. Kernel **execution template** (‘RAJA::kernel’)
2. Kernel **execution policies** (in ‘KERNEL_POL’)
3. Kernel **iteration spaces** (e.g., ‘RangeSegments’)
4. Kernel **body** (lambda expressions)

# Each loop level has an iteration space and loop variable

Nested loops

```
using namespace RAJA;
using KERNEL_POL = KernelPolicy<
                     statement::For<1, exec_policy_row,
                       statement::For<0, exec_policy_col,
                         statement::Lambda<0>
                       >
                     >
                   >;

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

  // ...
} );
```

The order (and types) of tuple items and lambda arguments must match.

# Each loop level has an execution policy

```
using namespace RAJA;
using KERNEL_POL = KernelPolicy<
                     statement::For<1, exec_policy_row,
                       statement::For<0, exec_policy_col,
                         statement::Lambda<0>
                       >
                     >
                   >;

RAJA::kernel<KERNEL_POL>( RAJA::make_tuple(col_range, row_range),
                          [=](int col, int row ) {
  // ...
} );
```

'For' statement integer parameter indicates tuple item it applies to: '0' → col, 1' → row.

# To transform the loop order, change the execution policy, not the kernel code

Nested loops

```
using KERNEL_POL = KernelPolicy<
                     statement::For<1, exec_policy_row,
                       statement::For<0, exec_policy_col,
                         ...
                     >;
```

Outer row loop (1), inner col loop (0)

'For' statements are swapped.

```
using KERNEL_POL = KernelPolicy<
                     statement::For<0, exec_policy_col,
                       statement::For<1, exec_policy_row,
                         ...
                     >;
```

Outer col loop (0), inner row loop (1)

This is analogous to swapping for-loops in a C-style implementation.

# RAJA::KernelPolicy constructs comprise a simple DSL that relies only on standard C++11 support

Nested loops

- A KernelPolicy is built from “Statements” and “StatementLists”
  - A **Statement is an action**: execute a loop, invoke a lambda, synchronize threads, etc. ,

```
For<0, exec_pol, ...>
```

```
Lambda<0>
```

```
CudaSyncThreads
```

  - A **StatementList is an ordered list of Statements** processed as a sequence; e.g.,

```
For<0, exec_policy0,
   Lambda<0>,
   For<2, exec_policy2,
      Lambda<1>
   >
>
```

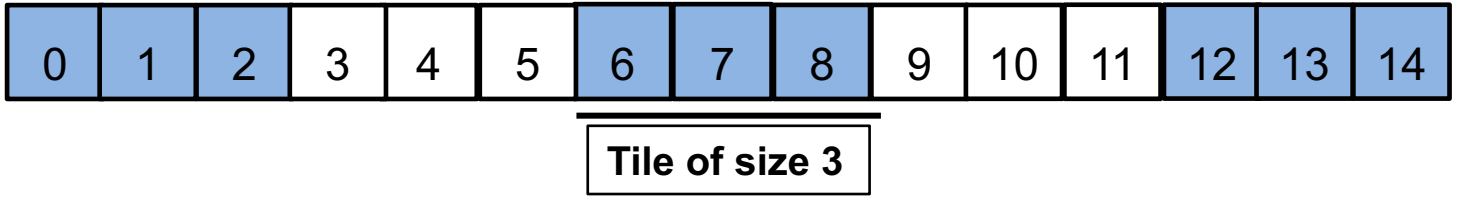
A RAJA::KernelPolicy type is a StatementList.

# RAJA provides various RAJA::statement types

- We discuss several of them in this presentation
- See the RAJA User Guide for a complete listing of available statement types and how they work

# Loop tiling

# With loop tiling, data is processed in chunks




```
// standard loop
for (int id = 0; id < N; ++id) {

}
```

```
// outer loop over tiles
for (int i = 0; i < N_tile; ++i) {

  // inner loop within a tile
  for (int ti = 0; ti < TILE_DIM; ++ti) {
    // compute global index
    int id = i * TILE_DIM + ti;
  }
}
```

This helps ensure data stays in cache while it is used; read-write overhead is reduced.
Tile size is a performance tuning parameter.

# Tiling can improve the performance of many algorithms

Loop tiling

- Computing a matrix transpose is an example
- Partition matrix into a set of tiles, then transpose data within each tile

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$A$

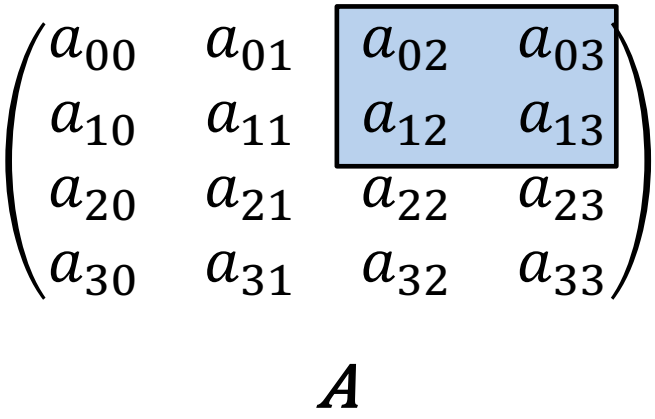
$$\begin{pmatrix} a_{00} & a_{10} & & \\ a_{01} & a_{11} & & \\ & & & \\ & & & \end{pmatrix}$$

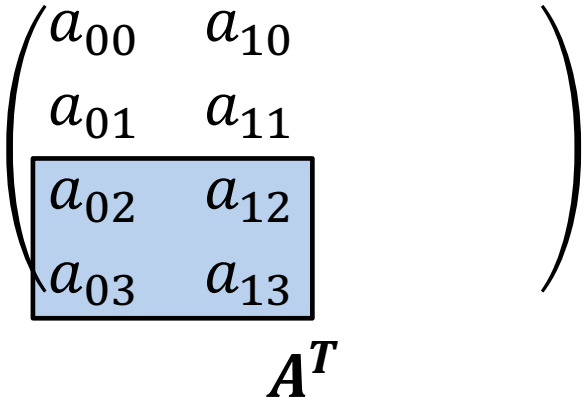
$A^T$

# Tiling can improve the performance of many algorithms

Loop tiling

- Done properly, this can improve spatial and temporal locality of data accesses




Tile data may be stored in CPU stack or GPU shared memory.

# C-style tiled matrix transpose operation without storing a local tile

Loop tiling

$A^T(c, r) = A(r, c)$, where A is $N_r$ x $N_c$ matrix and $A^T$ is $N_c$ x $N_r$ matrix

```
for (int br = 0; br < Ntile_r; ++br) {   // outer loops over tiles
  for (int bc = 0; bc < Ntile_c; ++bc) {

    for (int tr = 0; tr < TILE_SZ; ++tr) {  // inner loops within a tile
      for (int tc = 0; tc < TILE_SZ; ++tc) {

        int row = br * TILE_SZ + tr;  // global row index
        int col = bc * TILE_SZ + tc;  // global column index

        if (row < N_r && col < N_c) { At(col, row) = A(row, col); }

      }
    }
  }
}
```

Note: in general, bounds checks are needed to prevent indexing out of bounds.

# RAJA tiling statements eliminate the need for manual global index computation and bounds checks

Loop tiling

```
using namespace RAJA;

using KERNEL_POL =
  KernelPolicy<
    statement::Tile<0, statement::tile_fixed<TILE_SZ>, seq_exec, // tile rows
      statement::Tile<1, statement::tile_fixed<TILE_SZ>, seq_exec, // tile cols

        ...
      >
    >
  >;
```

‘Tile’ statement types indicate tile structure for each for loop.

# RAJA tiling statements eliminate need for manual global index computation and bounds checks

Loop tiling

```
using namespace RAJA;

using KERNEL_POL =
  KernelPolicy<
    statement::Tile<0, statement::tile_fixed<TILE_SZ>, seq_exec, // tile rows
      statement::Tile<1, statement::tile_fixed<TILE_SZ>, seq_exec, // tile cols

        statement::For<0, seq_exec, // rows within a tile
          statement::For<1, seq_exec, // cols within a tile

            statement::Lambda<0>  // lambda body is: At(col, row) = A(row, col)

          >
        >
      >
    >
  >;
```

Nested loop constructs inside tile statements are the same as for non-tiled loops.

Note that global indices are calculated for you and passed as lambda args.

# (Thread) local data

# Sometimes kernels require multiple lambdas to fully describe implementation

Local data

- Until now, we have mostly considered perfectly nested loops (loop nests with no intervening code between loops) and loop bodies involving exactly one lambda

- Again, recall the matrix multiplication example:

```
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;
  }
}
```

How can we write this as a RAJA kernel that is portable and allows other performance enhancing features?

# Use lambda statements to define intervening code between loops

Local data

```
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;

  }
}
```

```
RAJA::Kernel<
  For<0, exec_policy_row,
    For<1, exec_policy_col,
      Lambda<0>
      For<2, seq_exec,
        Lambda<1>
      >,
      Lambda<2>
    >
  >
>
```

Composing policies like this can help you do architecture-specific optimizations in a portable way.

# RAJA::kernel_param takes a tuple for thread-local data (scalars and/or kernel-local arrays)

Local data

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

  RAJA::make_tuple( (double)0.0 ),    // thread local data

  [=] ( int row, int col, int k, double& foo ) {
     // lambda body
  },

  [=] ( int row, int col, int k, double& bar ) {
     // lambda body
  },

  ...

);
```

Lambda arguments are iteration space variables (row, col, k) and thread-local variable.

Note: thread-local data is not named in the tuple, can be named anything in a lambda argument list.

# RAJA::kernel_param takes a tuple for thread-local variables and/or kernel-local arrays

Local data

```
RAJA::kernel_param < KERNEL_POL >(
  RAJA::make_tuple(row_range, col_range, dot_range),
  RAJA::make_tuple( (double)0.0 ),    // thread local variable for 'dot'

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

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

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

Note that all lambdas have same args here. RAJA has ways to be more specific.

Thread-local data can be passed by-value or by-reference to a lambda so it's value can be accessed and updated as needed.

# Policy example: collapse loops in an OpenMP parallel region

Local data

```
using KERNEL_POL =
  RAJA::KernelPolicy<
    statement::Collapse<RAJA::omp_parallel_collapse_exec,
                        RAJA::ArgList<0, 1>, // row, col

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

    >
  >;
```

This policy distributes iterations in loops ‘0’ and ‘1’ across CPU threads.

Local data

# Policy example: launch loops as a CUDA kernel

```
using KERNEL_POL =
  RAJA::KernelPolicy<
    statement::CudaKernel<
      statement::For<0, RAJA::cuda_block_x_loop,     // row
        statement::For<1, RAJA::cuda_thread_x_loop, // col

          statement::Lambda<0>,          // dot = 0.0
          statement::For<2, RAJA::seq_exec,
            statement::Lambda<1>         // dot += ...
          >,
          statement::Lambda<2>           // set C(row, col) = ...

        >
      >
    >
  >;
```

This policy distributes ‘row’ indices over CUDA thread blocks and ‘col’ indices over threads in each block.

# Back to the matrix transpose loop tiling example…

Local data

```
for (int br = 0; br < Ntile_r; ++br) {   // Outer loops over tiles
  for (int bc = 0; bc < Ntile_c; ++bc) {

    int Tile[TILE_SZ][TILE_SZ];

    for (int tr = 0; tr < TILE_SZ; ++tr) {      // Read a tile of 'A'
      for (int tc = 0; tc < TILE_SZ; ++tc) {
        if (row < N_r && col < N_c) { Tile[tr][tc] = A(row, col); }
      }
    }

    for (int tc = 0; tc < TILE_SZ; ++tc) {      // Write a tile of 'At'
      for (int tr = 0; tr < TILE_SZ; ++tr) {
        if (row < N_r && col < N_c) { At(col, row) = Tile[tr][tc]; }
      }
    }
// etc.
```

C-style 'tiled' loop nest

Using a local stack array improves memory access efficiency in a tile.

# Different parallel strategies have different access requirements for local data

Local data

```
Parfor (int br = 0; br < Ntile_r; ++br) {
  for (int bc = 0; bc < Ntile_c; ++bc) {

    // Thread-private array
    int Tile[TILE_DIM][TILE_DIM];

    for (int tr = 0; tr < TILE_DIM; ++tr) {
      for (int tc = 0; tc < TILE_DIM; ++tc) {
        Tile[tr][tc] = A(row, col);
      }
    }

    // ...
  }
}
```

When the outer loop is parallel, tile data should be private to each thread

```
for (int br = 0; br < Ntile_r; ++br) {
  for (int bc = 0; bc < Ntile_c; ++bc) {

    // Shared array
    int Tile[TILE_DIM][TILE_DIM];

    Parfor (int tr = 0; tr < TILE_DIM; ++tr) {
      for (int tc = 0; tc < TILE_DIM; ++tc) {
        Tile[tr][tc] = A(row, col);
      }
    }

    // ...
  }
}
```

When an inner loop is parallel, tile data should be shared between threads

# RAJA provides a LocalArray type to help manage these cases in a portable manner

Local data

```
using namespace RAJA;
using TILE_MEM = LocalArray<int, Perm<0, 1>, SizeList<TILE_SZ, TILE_SZ>>;

TILE_MEM TileArray;
```

The LocalArray type defines a multi-dimensional array of fixed size that can be used in a kernel.

# A local array object is allocated inside a kernel

```
using namespace RAJA;
using TILE_MEM = LocalArray<int, Perm<0, 1>, SizeList<TILE_SZ, TILE_SZ>>;

TILE_MEM TileArray;

using EXEC_POL = KernelPolicy<
  statement::Tile<0, statement::tile_fixed<TILE_SZ>, loop_exec,
    statement::Tile<1, statement::tile_fixed<TILE_SZ>, loop_exec,

      statement::InitLocalMem<tile_mem_policy, ParamList< # >,
        . . .
        >
    >
  >
>;
```

The local array is allocated for use in a kernel using the 'InitLocalMem' statement. The initialization requires a memory policy and binds the local array object to a slot in the parameter tuple (#).

# The local array can be accessed in any lambda in the kernel

Local data

```
using namespace RAJA;
using TILE_MEM = LocalArray<int, Perm<0, 1>, SizeList<TILE_SZ, TILE_SZ>>;
TILE_MEM TileArray;

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

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

  [=](int row, int col, int tr, int tc, TILE_MEM& TileArray) {
    TileArray(tc, tr) = Aview(row, col);
  },

  [=](int row, int col, int tr, int tc, TILE_MEM TileArray) {
    Atview(col, row) = TileArray(tc, tr);
  }
);
```

Local indices tx, ty are first two entries in param tuple

Lambda args:
- Global indices
- Local tile indices
- LocalArray for tile data

# RAJA provides memory policy types for different local array data

Local data

**`RAJA::cpu_tile_mem`** – Use CPU stack memory

**`RAJA::cuda_shared_mem`** – Use CUDA shared memory (sharable across threads in a CUDA thread block)

**`RAJA::cuda_thread_mem`** – Use memory local to a CUDA thread

# Application considerations

# “Bring your own” memory management

- RAJA does not provide a memory model. This is by design.
  - Users must handle memory space allocations and transfers

# “Bring your own” memory management

- RAJA does not provide a memory model…..by design
  - Users must handle memory space allocations and transfers

```
forall<cuda_exec>(range, [=] __device__ (int i) {
  a[i] = b[i];
} );
```

Are ‘a’ and ‘b’ accessible on GPU?

# “Bring your own” memory management

- RAJA does not provide a memory model…..by design
  - Users must handle memory space allocations and transfers

```
forall<cuda_exec>(range, [=] __device__ (int i) {
  a[i] = b[i];
} );
```

Are ‘a’ and ‘b’ accessible on GPU?

- Some possibilities:
  - **Manual** – use cudaMalloc( ), cudaMemcpy( ) to allocate, copy to/from device
  - **Unified Memory (UM)** – use cudaMallocManaged( ), paging on demand
  - **CHAI** (https://github.com/LLNL/CHAI) – automatic data copies as needed

CHAI was developed to complement RAJA.

# RAJA promotes flexibility via type parameterization

- Define **type aliases in header files**
  - Easy to explore implementation choices in a large code base
  - Reduces source code disruption

# RAJA promotes flexibility and tuning via type parameterization



- Define **type aliases in header files**
  - Easy to explore implementation choices in a large code base
  - Reduces source code disruption
- Assign execution policies to **"loop/kernel classes"**
  - Easier to search execution policy parameter space

```
using ELEM_LOOP_POLICY = ...; // in header file

RAJA::forall<ELEM_LOOP_POLICY>( /* do elem stuff */ );
```

Application developers must determine the "loop taxonomy" and policy selection for their code.

# Performance portability takes effort

- RAJA (like any programming model) is an enabling technology – not a panacea
  - Achieving and maintaining thread safety in kernels is critical
  - Loop characterization and performance tuning are manual processes
    - Good tools are essential!!
  - Memory motion and access patterns are critical. Pay attention to them!
    - True for CPU code as well as GPU code

# Performance portability takes effort

- Application coding styles may need to change regardless of programming model for fine-grained loop-level parallelism
  - Change algorithms as needed to ensure correct parallel execution
  - Move variable declarations to innermost scope to avoid threading issues
  - Recast some patterns as reductions, scans, etc.
  - Virtual functions and C++ STL are problematic for GPU execution

Simpler is almost always better – use simple types and arrays for GPU kernels.

# Data dependencies are a key inhibitor of parallelism

Apps

- A data dependence occurs when multiple threads or tasks could **write to the same memory location at the same time – “race condition” → non-deterministic results**

# Data dependencies are a key inhibitor of parallelism



- A data dependence occurs when multiple threads or tasks could write to the same memory location at the same time – "race condition" → non-deterministic results
  - For example: a loop where iterations are **not** independent

```
for (int i = 0; i < N; ++i) {
   x[i] = x[i-1] + y[i];
}
```

```
for (int r = 0; r < N; ++r) {
   for (int c = 0; c < N; ++c) {
      A[r][c] = A[c][r];
   }
}
```

```
double sum = 0.0;
for (int i = 0; i < N; ++i) {
   sum += a[i];
}
```

VS.

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

"data parallel" loop

# Data dependencies are a key inhibitor of parallelism



- A data dependence occurs when multiple threads or tasks could write to the same memory location at the same time – "race condition" → non-deterministic results
  - For example: a loop where iterations are **not** independent

```
for (int i = 0; i < N; ++i) {
   x[i] = x[i-1] + y[i];
}
```

```
for (int r = 0; r < N; ++r) {
   for (int c = 0; c < N; ++c) {
      A[r][c] = A[c][r];
   }
}
```

```
double sum = 0.0;
for (int i = 0; i < N; ++i) {
   sum += a[i];
}
```

VS.

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

"data parallel" loop

Sometimes algorithms must be rewritten to enable parallelism.

# Amdahl's law tells us speedup is always limited by sequential portions of your code

Apps

- Amdahl's law:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

$S(n)$ is the theoretical maximum speedup of a workload run in parallel with *n* processors

$p$ is the proportion of sequential run time (1 processor) of the parts of the program that can run in parallel

# Amdahl's law tells us speedup is always limited by sequential portions of your code

Apps

- Amdahl's law:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

$S(n)$ is the theoretical maximum speedup of a workload run in parallel with *n* processors

$p$ is the proportion of sequential run time (1 processor) of the parts of the program that can run in parallel

- Note the following:

$$S(n) \leq \frac{1}{(1-p)} \qquad \lim_{n\to\infty} S(n) = \frac{1}{(1-p)}$$

"p" is the limiting factor.

# How do you know what you actually gain from parallelism?

Apps

- Measure and compare sequential run time and parallel run time
  - "Parallel Speedup" $S_n = T_1 / T_n$
  - "Parallel Efficiency" $E_n = S_n / n$

$T_1$ is sequential run time

$T_n$ is run time using n processes or threads

$S_n = n$ ($E_n = 1$) ⇒ "Perfect (ideal) scaling"

Most of the time, you will see $S_n < n$ ($E_n < 1$). However, it is possible to see $S_n > n$.

# Reducing memory motion is critical for good performance

Apps

- General “rules of thumb”
  - Place data that are **used together close in memory**: think locality – spatial and temporal
  - Consider **data access patterns** when designing algorithms

# Reducing memory motion is critical for good performance

Apps

- General “rules of thumb”
  - Place data that are used together close in memory: think locality – spatial and temporal
  - Consider data access patte**rns** when designing algorithms
- **Memory coalescing** is very important for GPU performance
  - Multiple memory accesses are combined into one memory transaction
  - With CUDA, you typically want all 32 threads in a warp to read operands & write results in as few transactions as possible and avoid serialized memory access
  - **Avoid memory accesses that are non-sequential, sparse, or misaligned**

# Reducing memory motion is critical for good performance

Apps

- General “rules of thumb”
  - Place data that are used together close in memory: think locality – spatial and temporal
  - Consider data access patte**rns** when designing algorithms
- **Memory coalescing** is very important for GPU performance
  - Multiple memory accesses are combined into one memory transaction
  - With CUDA, you typically want all 32 threads in a warp to read operands & write results in as few transactions as possible and avoid serialized memory access
  - **Avoid memory accesses that are non-sequential, sparse, or misaligned**
- Useful references:
  - “What Every Programmer Should Know About Memory” by Ulrich Drepper (https://akkadia.org/drepper/cpumemory.pdf)
  - “Introduction to GPGPU and CUDA Programming” by Philip Nee (https://cvw.cac.cornell.edu/GPU/default)

# Wrap-up

# Consider your application's characteristics and constraints when deciding how to use RAJA in it



- Profile your code to see where performance is most important
  - Do a few kernels dominate runtime?
  - Does no subset of kernels take a significant fraction of runtime?
  - Can you afford to maintain multiple, (highly-optimized) architecture-specific versions of important kernels?
  - Do you require a truly portable, single-source implementation?

# Consider your application's characteristics and constraints when deciding how to use RAJA in it



- Construct a taxonomy of algorithm patterns/loop structures in your code
  - Is it amenable to grouping into classes of RAJA usage (e.g., execution policies) so that you can propagate changes throughout the code base easily with header file changes?
  - If you have a large code with many kernels, it will be easier to port to RAJA if you define policy types in a header file and apply each to many loops

# Consider your application's characteristics and constraints when deciding how to use RAJA in it



- Consider developing a lightweight wrapper layer around RAJA
  - How important is it that you preserve the look and feel of your code?
  - How comfortable is your team with software disruption and using C++ templates?
  - Is it important that you limit implementation details to your CS/performance tuning experts?

# Materials that supplement this presentation are available

- Complete working example codes are available in the RAJA source repository
  - https://github.com/LLNL/RAJA
  - Many similar to examples we presented today and expands on them
  - Look in the “RAJA/examples” and “RAJA/exercises” directories
- The RAJA User Guide
  - Topics we discussed today, plus configuring & building RAJA, etc.
  - Available at http://raja.readthedocs.org/projects/raja (also linked on the RAJA GitHub project)

# Related software is also available

- The RAJA Performance Suite
  - Algorithm kernels in RAJA and baseline (non-RAJA) forms
  - Sequential, OpenMP (CPU), OpenMP target, CUDA variants
  - We use it to monitor RAJA performance and assess compilers
  - Essential for our interactions with vendors
  - Benchmark for CORAL and CORAL-2 systems
  - https://github.com/LLNL/RAJAPerf

# More related software…

- CHAI
  - Provides automatic data copies to different memory spaces behind an array-style interface
  - Designed to work with RAJA
  - Could be used with other lambda-based C++ abstractions
  - https://github.com/LLNL/CHAI

# Again, we would appreciate your feedback…

- If you have comments, questions, suggestions, etc., please talk to one of us
- You are welcome to join our Google Group linked to our Github repository home page (https://github.com/LLNL/RAJA)
- Or contact us via our team email list: raja-dev@llnl.gov

# Thank you for your attention and participation

## Questions?