A Tutorial Introduction to RAJA

RAJA

EXASCALE COMPUTING PROJECT

August, 2022

RAJA Team
Presented by Robert Chen

LLNL-PRES-838143

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 to the RAJA tutorial

- Today, we will describe RAJA and how it enables performance portability
- We will also present some background material that may help you think about key issues when developing parallel applications
- We will present examples that show you how to use RAJA
- Our objective for today is to teach you enough to start using RAJA in your own code development

See the RAJA User Guide for more information (readthedocs.org/projects/raja/).

During the tutorial…

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 us a message to our project email list: raja-dev@llnl.gov
- We appreciate specific, concrete feedback that helps us improve RAJA and this tutorial

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, GPUs (NVIDIA, AMD, Intel), …
- RAJA **insulates application source code** from hardware and programming model-specific implementation details
  - OpenMP, CUDA, HIP, SIMD vectorization, …
- RAJA supports a variety of **parallel patterns** and **performance tuning** options
  - Simple and complex loop kernels
  - Reductions, scans, sorts, 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**

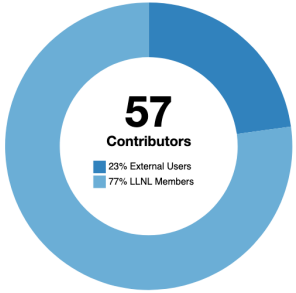
These goals have been affirmed by production application teams using RAJA.

RAJA is an open-source project with a growing user and contributor base

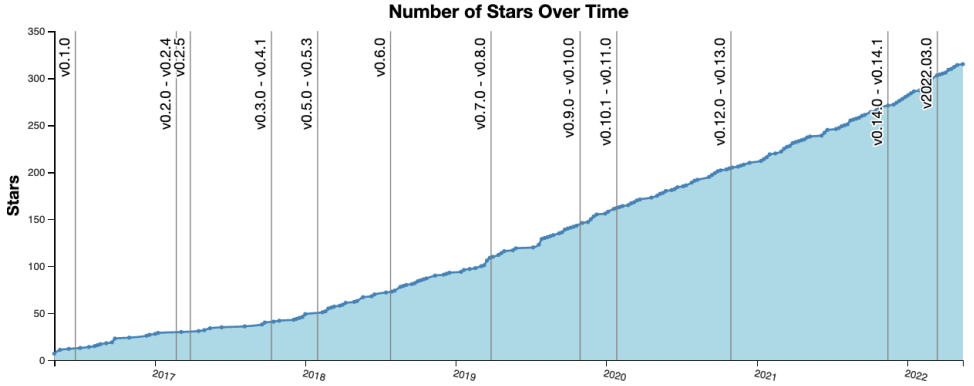
RAJA

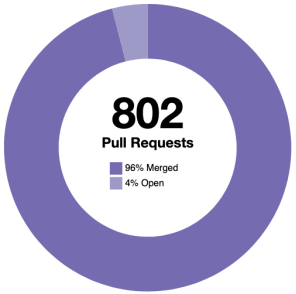
LLNL | C++ | BSD-3-Clause

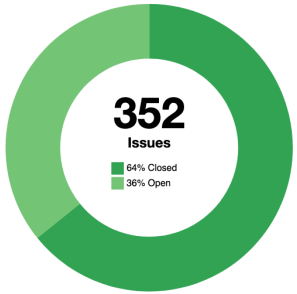
GitHub Page Stargazers : 320 Forks : 80



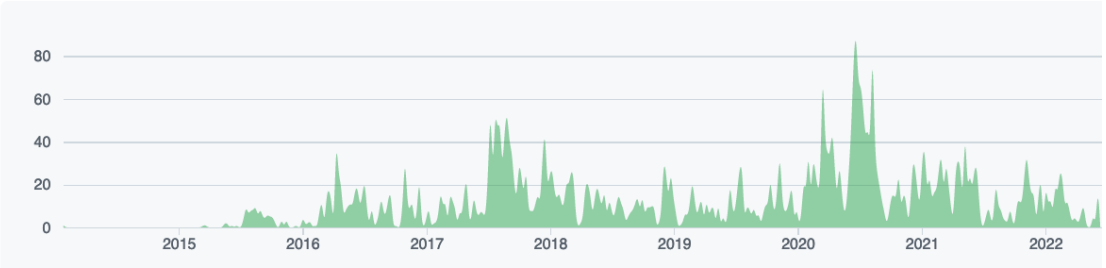
Project stats on 6/17/2022







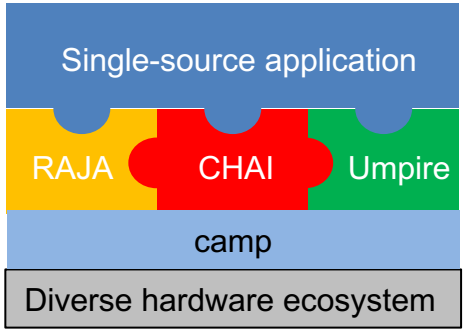
Contributions to develop, excluding merge commits and bot accounts



RAJA is part of the RAJA Portability Suite, which contains four complementary projects

RAJA: C++ kernel execution abstractions

Enable single-source application code insulated from hardware and programming model details



Umpire: Memory management API

High performance memory operations, such as pool allocations, with native C++, C, Fortran APIs

camp: C++ metaprogramming facilities

Focuses on HPC compiler compatibility and portability

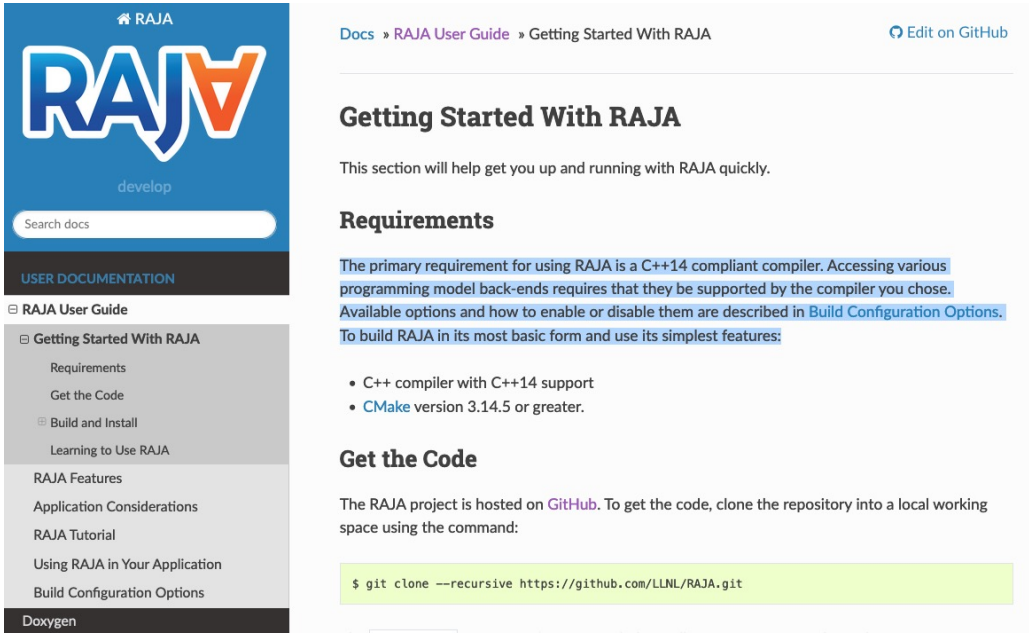
https://github.com/LLNL/RAJA
https://github.com/LLNL/CHAI
https://github.com/LLNL/Umpire
https://github.com/LLNL/camp

CHAI: C++ array abstractions

Automates data copies, based on RAJA execution contexts, giving apps the look and feel of unified memory, but with better performance

URLs with useful information about RAJA…

- **RAJA User Guide:** getting started info, details about features and usage, etc. (readthedocs.org/projects/raja)
- **RAJA Project Template:** shows how to use RAJA and BLT in an application that uses CMake (https://github.com/LLNL/RAJA-project-template)
- **RAJA Proxy Apps:** a collection of proxy apps written using RAJA (https://github.com/LLNL/RAJAProxies)
- **RAJA Performance Suite:** a large collection of loop kernels for assessing compilers and RAJA performance. Used by us, vendors, for DOE platform procurements, etc. (https://github.com/LLNL/RAJAPerf)



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

We will cover various topics today

- RAJA usage considerations (C++ templates, lambdas, memory management, etc.)
- Generally useful information for reasoning about parallel algorithms
- RAJA features:
  - **Simple loops**
  - **Reductions**
  - **Iteration spaces**
  - **Data layouts and views**
  - **Complex loop kernels**
  - *Atomic, scan and sort operations (briefly)*
  - *More advanced features of the RAJA Portability Suite (briefly)*

Let's start simple…

Simple loop execution

Consider a simple 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: all aspects of execution are explicit in the source code – execution (sequential), loop iteration order, data access pattern, etc.

RAJA encapsulates loop execution details

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];
}
```

Definitions like these are typically placed in headers where they can be easily reused throughout the source code.

RAJA-style loop

```
using EXEC_POL = ...;

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

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::TypedRangeSegment<int> 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::TypedRangeSegment<int> 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., ‘TypedRangeSegment’)
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 (stride-1 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 safety

The execution policy determines the programming model back-end


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

```
RAJA::loop_exec
```

```
RAJA::omp_parallel_for_exec
```

```
RAJA::cuda_exec<BLOCK_SIZE>
```

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

```
RAJA::tbb_for_exec
```

A sampling of RAJA loop execution policy types.

RAJA supports a variety of kernel execution mechanisms...

- Sequential (forces strictly sequential execution)
- “Loop” (lets compiler decide which optimizations to apply)
- OpenMP multithreading (CPU)
- TBB (Intel Threading Building Blocks) – partial support
- CUDA (NVIDIA GPUs)
- HIP (AMD GPUs)
- SYCL (Intel GPUs) – work-in-progress
- ”Vectorization” – SIMD (CPU), tensor & matrix cores (GPU)
- OpenMP target (available target device; e.g., GPU) – not considered production quality

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 a few RAJA usage considerations

RAJA makes heavy use of C++ templates

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

- Templates allow one to write *generic* code and have the *compiler generate* a specific implementation for each set of given template parameter types
- 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) {
   ...
}
```

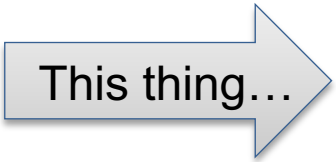
- "ExecPol", "IdxType", "LoopBody" are C++ types you specify

Like this…

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

- Note: "IdxType" and "LoopBody" types are deduced by the compiler based on your code

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




```
forall<seq_exec>(TypedRangeSegment<int>(0, N),
  [=] (int i) {
     x[i] = a * x[i] + y[i];
  }
);
```

- 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>(TypedRangeSegment<int>(0, N), [=] (int i) {
    x[i] = a * x[i] + y[i];
});
```

- A lambda expression has the following form

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

Lambda expression concepts…

```
forall<seq_exec>(TypedRangeSegment<int>(0, N), [=] (int i) {
    x[i] = a * x[i] + y[i];
});
```

- 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
  - Value or reference (**[=]** vs. **[&]**)? By-value is required for GPU execution, RAJA reductions, etc.
  - **We recommend using capture by-value in all cases**, as shown above

Lambda expression concepts…

```
forall<seq_exec>(TypedRangeSegment<int>(0, N), [=] (int i) {
    x[i] = a * x[i] + y[i];
});
```

- 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 – (**`int i`**)

Lambda expression concepts…

```
forall<seq_exec>(TypedRangeSegment<int>(0, N), [=] (int i) {
    x[i] = a * x[i] + y[i];
});
```

- 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; e.g., (**`int i`**)
- A lambda passed to a CUDA or HIP kernel requires a *device annotation*:

[=] **__device__** (…) { … }

Lambda expression concepts…

```
forall<seq_exec>(TypedRangeSegment<int>(0, N), [=] (int i) {
    x[i] = a * x[i] + y[i];
});
```

- 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.

“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. This is 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];
} );
```

‘a’ and ‘b’ must be accessible on GPU!!

- Some possibilities for getting data into GPU memory:
  - **Manual** – e.g., use cudaMalloc( ), cudaMemcpy( ) to allocate, copy to/from device
  - **Unified Memory (UM)** – e.g., use cudaMallocManaged( ), paging on demand
  - **Umpire** – a common interface regardless of system and memory type
  - **CHAI** – automatic data copies as needed

CHAI and Umpire are part of the RAJA Portability Suite.

“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];
} );
```

‘a’ and ‘b’ must be accessible on GPU!!

For simplicity, all RAJA exercises and examples in the repository use unified memory or manual copies for GPU versions.

Simple Forall Quiz!

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** in the latest release contains a complete implementation of this quiz.

C-style

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

Which RAJA execution policy emulates the C program?

seq_exec
loop_exec
omp_parallel_for_exec
cuda_exec

RAJA-version

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

Simple Forall Quiz!

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** in the latest release contains a complete implementation of this quiz.

C-style

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

Which RAJA execution policy emulates the C program?

~~seq_exec~~
~~loop_exec~~
omp_parallel_for_exec
~~cuda_exec~~

RAJA-version

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

Simple Forall Quiz!

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** in the latest release contains a complete implementation of this quiz.

C-style

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

Which RAJA execution policy emulates the C program?

`seq_exec`
`loop_exec`
`omp_parallel_for_exec`
`cuda_exec`

RAJA-version

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

Simple Forall Quiz!

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** in the latest release contains a complete implementation of this quiz.

C-style

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

Which RAJA execution policy emulates the C program?

~~seq_exec~~

loop_exec

~~omp_parallel_for_exec~~

~~cuda_exec~~

RAJA-version

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

loop_exec lets the compiler vectorize if possible.

seq_exec executes sequentially using `#pragma no vector`.

Simple Forall Quiz!

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** in the latest release contains a complete implementation of this quiz.

C-style

```
__global__ void addvec(double* c, double* a, double* b, N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) { c[i] = a[i] + b[i]; }
}
addvec<<< grid_size, block_size >>>( c, a, b, N );
```

cuda_exec
Note: Need to specify number of threads per block as a template parameter of the policy.

RAJA-version

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

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];
}
```

RAJA reduction objects hide the complexity of parallel reduction operations

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 reduction value inside a kernel. Different threads would see different partial reduction values since synchronization happens after the kernel completes.

- 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;
```

Sample RAJA reduction policy types.

RAJA supports five common reductions types

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

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

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

RAJA::ReduceMinLoc< REDUCE_POLICY, DTYPE > r(val_init,
                                             loc_init);

RAJA::ReduceMaxLoc< REDUCE_POLICY, DTYPE > r(val_init,
                                             loc_init);
```

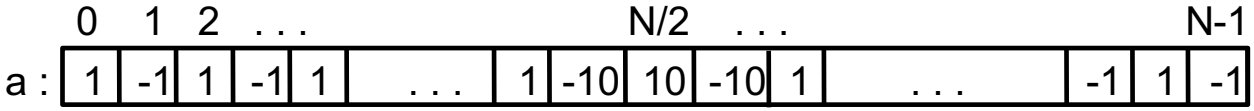
"Loc" reductions give a loop index where reduced value was found.

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::TypedRangeSegment<int>(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 setup…

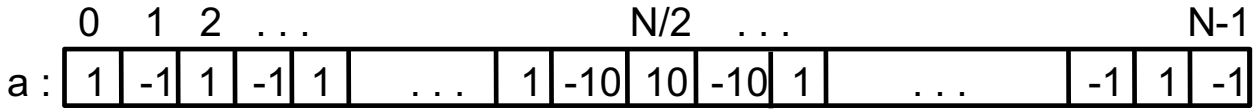
‘a’ is an int vector of length ‘N’ (N / 2 is even) initialized as:



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

Suppose we run the code on the previous slide with this setup…

‘a’ is an int vector of length ‘N’ (N / 2 is even) initialized as:



- *What are the reduced values?*
  - Sum = -9
  - Min = -10
  - Max = 10
  - Max-loc = N/2
  - Min-loc = N/2 – 1 or N/2 + 1 (order-dependent)

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

Reduction Quiz!

What is the value of z?

RAJA

```
RAJA::ReduceSum< RAJA::omp_reduce, int > y(1);

RAJA::forall< RAJA::omp_parallel_for_exec >(
  RAJA::TypedRangeSegment<int>(0, 4),
  [=] (int i) {
    y += i * 2;
  }
);
int z = y.get() + 3;
```

Reduction Quiz!

What is the value of z?

RAJA

```
RAJA::ReduceSum< RAJA::omp_reduce, int > y(1);

RAJA::forall< RAJA::omp_parallel_for_exec >(
  RAJA::TypedRangeSegment<int>(0, 4),
  [=] (int i) {
    y += i * 2;
  }
);
int z = y.get() + 3;
```

z = 1 + 0*2 + 1*2 + 2*2 + 3*2 + 3 = 16
(y is initialized to 1)

Reduction Quiz!

What is the value of z?

RAJA

```
RAJA::ReduceSum< RAJA::omp_reduce, int > y(1);

RAJA::forall< RAJA::omp_parallel_for_exec >(
  RAJA::TypedRangeSegment<int>(0, 4),
  [=] (int i) {
    y += i * 2;
  }
);
int z = y.get() + 3;
```

C-style

```
int z = 1;
#pragma omp parallel for reduction(+:z)

for (int i = 0; i < 4; ++i) {
  z += i * 2;
}
z += 3;
```

z = 1 + 0*2 + 1*2 + 2*2 + 3*2 + 3 = 16
(y is initialized to 1)

Reduction Quiz!

What is the value of z?

RAJA

```
RAJA::ReduceSum< RAJA::omp_reduce, int > y(1);

RAJA::forall< RAJA::omp_parallel_for_exec >(
  RAJA::TypedRangeSegment<int>(0, 4),
  [=] (int i) {
    y += i * 2;
  }
);
int z = y.get() + 3;
```

C-style

```
int z = 1;
#pragma omp parallel for reduction(+:z)

for (int i = 0; i < 4; ++i) {
  z += i * 2;
}
z += 3;
```

z = 1 + 0*2 + 1*2 + 2*2 + 3*2 + 3 = 16
(y is initialized to 1)

The sequential and CUDA RAJA variants look the same, except for the policies . . .

RAJA::seq_exec – RAJA::seq_reduce

RAJA::cuda_exec<threads> – RAJA::cuda_reduce

Iteration spaces : Segments and IndexSets

A RAJA “Segment” defines a loop iteration space

Iteration spaces

- A **Segment** defines a set of loop indices to run in a kernel

Contiguous range [beg, end)

Strided range [beg, end, stride)

List of indices (indirection)

Loop iteration spaces are defined by Segments

Iteration spaces

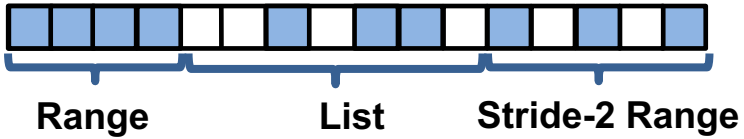
- A Segment defines a set of loop indices to run in a kernel

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 RAJA loop execution template.

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

Iteration spaces

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

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

Runs DAXPY loop indices: 0, 1, 2, … , N-1

A RangeStrideSegment defines a strided sequence of indices

Iteration spaces

```
RAJA::TypedRangeStrideSegment<int> srange1( 0, N, 2 );

RAJA::forall< RAJA::seq_exec >( srange1 , [=] (int i)
{
    x[i] = a * x[i] + y[i];
} );
```

Runs DAXPY loop indices: 0, 2, 4, …

A RangeStrideSegment defines a strided sequence of indices

Iteration spaces

```
RAJA::TypedRangeStrideSegment<int> srange1( 0, N, 2 );

RAJA::forall< RAJA::seq_exec >( srange1 , [=] (int i) {
    x[i] = a * x[i] + y[i];
} );
```

How do we get the odd indices? 1, 3, 5, …

```
RAJA::TypedRangeStrideSegment<int> srange2( ?, N, ? );

RAJA::forall< RAJA::seq_exec >( srange2 , [=] (int i) {
    x[i] = a * x[i] + y[i];
} );
```

A RangeStrideSegment defines a strided sequence of indices

Iteration spaces

```
RAJA::TypedRangeStrideSegment<int> srange1( 0, N, 2 );

RAJA::forall< RAJA::seq_exec >( srange1 , [=] (int i) {
    x[i] = a * x[i] + y[i];
} );
```

How do we get the odd indices? 1, 3, 5, …

```
RAJA::TypedRangeStrideSegment<int> srange2( 1, N, 2 );

RAJA::forall< RAJA::seq_exec >( srange2 , [=] (int i) {
    x[i] = a * x[i] + y[i];
} );
```

RangeStrideSegments also support negative indices and strides

Iteration spaces

```
RAJA::TypedRangeStrideSegment<int> srange3( N-1, -1, -1 );

RAJA::forall< RAJA::seq_exec >( srange3 , [=] (int i) {
    x[i] = a * x[i] + y[i];
} );
```

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

A ListSegment can define any set of indices

Iteration spaces

```
using IdxType = int;
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 = int;
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] = ...;
} );
```

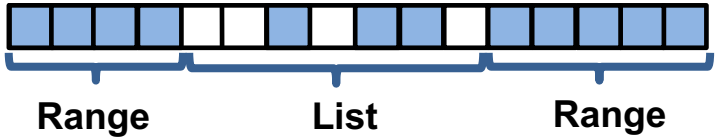
Runs loop indices: 10, 11, 14, 20, 22

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

IndexSets enable iteration space partitioning

Iteration spaces

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

RangeSegType range1(0, 8);

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

RangeSegType range3(24, 28);
```



IndexSets (e.g. **iset**) can be passed to RAJA kernel iteration methods to execute the collection of segments in one call.

```
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

Views and Layouts

Matrices and tensors are ubiquitous in scientific computing

View/Layout

- They are most naturally 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) {

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

   }
}
```

C-style matrix multiplication

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

RAJA Views and Layouts simplify multi-dimensional indexing

View/Layout

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

```
double* A = new double[ N * N ];

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

RAJA Views and Layouts simplify multi-dimensional indexing

View/Layout

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

```
double* A = new double[ N * N ];

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

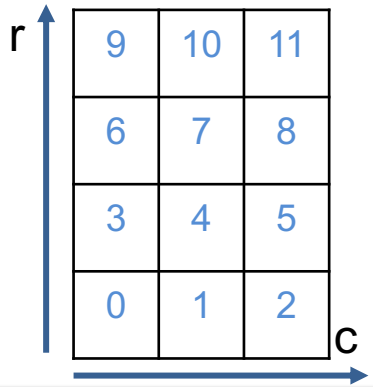
- This leads to data indexing that is simpler, more intuitive, and less error-prone

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

The RAJA default layout uses ‘row-major’ ordering (C/C++ standard convention).
So, the right-most index is stride-1 when using the Layout<DIM>.

Every Layout has a permutation

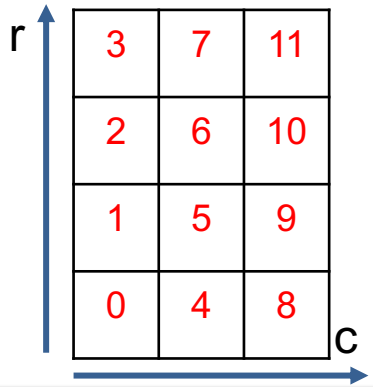
```
std::array<RAJA::idx_t, 2> perm {{0, 1}};  // default permutation

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 is stride-1 (rightmost in permutation). This follows the C++ specification of row-major memory indexing.

Every Layout has a permutation

```
std::array<RAJA::idx_t, 2> perm {{1, 0}};  // alternate 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) = ...;
```



"r" index is stride-1
(leftmost in permutation).

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

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

An offset layout applies an offset to indices

```
double* C = new double[10];

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 < 5; ++i) {
  CView(i) = ...;
}
```

A 1-d View with index offset and extent 10 [-5, 5).

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

Offset layouts are useful for index space subset operations (e.g., halo regions).

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?*

The right-most index 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?*

The right-most index is stride-1 (using default permutation).

- *What is the stride of the left-most index?*

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?*

The right-most index is stride-1 (using default permutation).

- *What is the stride of the left-most index?*

It has stride 10 (since the right-most index has extent 10, [-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).

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).

- *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).

- *Which index is stride-1?*

The **left-most** index 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).

- *Which index is stride-1?*

The **left-most** index has stride-1 (due to permutation).

- *What is the stride of the right-most index?*

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).

- *Which index is stride-1?*

The **left-most** index has stride-1 (due to permutation).

- *What is the stride of the right-most index?*

The right-most index has stride 3 (since the left-most index has extent 3, [-1, 2)).

Complex Loops and Advanced RAJA Features

Nested Loops with RAJA Kernel API

We will use a nested loop kernel for matrix multiplication to explore RAJA features and usage

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;

  }
}
```

The RAJA *kernel* API is designed to compose and transform complex parallel kernels

Nested loops

```
using namespace RAJA;
using KERNEL_POL = KernelPolicy<
                     statement::For<1, row_policy,
                       statement::For<0, col_policy,
                         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 in C-style version.

Each loop level has an execution policy and iteration space

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

  // row-column dot product

  }
}
```

```
using EXEC_POL = KernelPolicy<
  statement::For<1, row_policy,
    statement::For<0, col_policy,
      statement::Lambda<0>
    >
  >
>;

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

    // row-column dot product

});
```

Integer parameter in each ‘For’ statement indicates the iteration space tuple item it applies to.

Kernel transformations are made by altering the execution policy, not the algorithm source code

Nested loops

```
using EXEC_POL = KernelPolicy<
  statement::For<1, row_policy,
    statement::For<0, col_policy,
      ...
>;
```

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

'For' statements are swapped.

```
using EXEC_POL = KernelPolicy<
  statement::For<0, col_policy,
    statement::For<1, row_policy,
      ...
>;
```

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

This is analogous to swapping for-loops in the C-style version.

Nested loops

Lambda statements invoke lambda expressions (e.g. loop bodies)

```
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;

  }
}
```

```
using EXEC_POL = KernelPolicy<
  statement::For<1, row_policy,
    statement::For<0, col_policy,

      RAJA::statement::Lambda<0>

    >
  >
>;

kernel<EXEC_POL>(
  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;

});
```

Nested Loops with RAJA *Launch* API

The RAJA launch API creates an execution space for writing nested loops using RAJA loop methods

Nested loops

```
launch<launch_policy>(
  Grid(Teams(NTeams), Threads(NThreads)))
  [=] RAJA_HOST_DEVICE (LaunchContext ctx) {

  loop<row_policy>(ctx, row_range, [&](int row){
    loop<col_policy>(ctx, 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;
    });
  });

});
```

Kernel execution space

Methods and types in RAJA::expt namespace

Loops are expressed inside the execution space using RAJA loop methods

Nested 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(row, k)* B(k, col)
   }

   C(row, col) = dot;

  }
}
```

```
launch<launch_policy>(
  Grid(Teams(NTeams), Threads(NThreads)))
  [=] RAJA_HOST_DEVICE(LaunchContext ctx) {

  loop<row_policy>(ctx, row_range, [&](int row){
    loop<col_policy>(ctx, 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;

  });
});

});
```

Methods and types in RAJA::expt namespace

GPU execution using RAJA launch uses a thread/team model equivalent to the CUDA/HIP thread/block model

Nested loops

```
launch<launch_policy>(
  Grid(Teams(NTeams), Threads(NThreads)))
  [=] RAJA_HOST_DEVICE (LaunchContext ctx) {

  loop<row_policy>(ctx, row_range, [&](int row){
    loop<col_policy>(ctx, 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;
    });
  });
});
```

Teams = CUDA/HIP Blocks
Threads = CUDA/HIP Threads

Loops can be mapped to CUDA/HIP threads or blocks

Runtime in RAJA launch/loop methods is made possible when providing both host and device policies

Nested loops

```
using launch_policy =
LaunchPolicy<host_launch_t, device_launch_t>

using row_policy =
 LoopPolicy<host_policy, device_policy>;

using col_policy =
LoopPolicy<host_policy, device_policy>;
```

- Supported host backends: Sequential, OpenMP
- Supported device backends: CUDA/HIP

`cpu_or_gpu` represents runtime choice of host or device execution. This is optional if one policy is provided.

```
launch<launch_policy>(cpu_or_gpu,
  Grid(Teams(NTeams), Threads(NThreads)))
  [=] RAJA_HOST_DEVICE (LaunchContext ctx) {

  loop<row_policy>(ctx, row_range, [&](int row){
    loop<col_policy>(ctx, row_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;
    });
  });
});
```

Methods and types in RAJA::expt namespace

CUDA's hierarchical parallelism can be expressed as nested for loops inside the RAJA launch method

Nested loops

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

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

   C(row, col) = dot;

}
```

Matrix-Matrix multiplication kernel

```
using row_policy =
LoopPolicy<host_policy, cuda_block_x_direct>;

using col_policy =
LoopPolicy<host_policy, cuda_thread_x_loop>;
                    . . .
loop<row_policy>(ctx, row_range, [&](int row){
   loop<col_policy>(ctx, 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;
  });
});
```

- Runtime for N = 1e4 on NVIDIA V100: 3793 milliseconds

- Runtime for N = 1e4 on NVIDIA V100: 2921 milliseconds

Nested loops

Global thread ID calculations are simplified with RAJA policies

```
int row =
blockIdx.y * blockDim.y + threadIdx.y;

int col =
 blockIdx.x * blockDim.x +
threadIdx.x;

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

Matrix-Matrix multiplication kernel with global threads

- Runtime for N = 1e4 on NVIDIA V100 : 1297 milliseconds

```
using row_policy =
LoopPolicy<loop_exec, cuda_global_thread_y>;

using col_policy =
LoopPolicy<loop_exec, cuda_global_thread_x>;

. . .
loop<row_policy>(ctx, row_range, [&](int row) {
  loop<col_policy>(ctx, 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;
  });
});
```

- Runtime for N = 1e4 on NVIDIA V100 : 1313 milliseconds (within 2%)

Brief Overview of Atomics, Scan, Sort

Atomics: RAJA OpenMP 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;

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

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

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

`pi` may be simultaneously written by multiple threads during the `forall` loop. The `atomicAdd` operation on `pi` ensures locked access; each thread has exclusive serialized write access.

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

Scan: RAJA provides a default prefix-sum scan operation

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

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

'in' and 'out' are input and output arrays of length N, respectively. 'out' holds partial sums of the input array.

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 "+".

A 3rd argument can be given to specify a scan operator, e.g. `RAJA::operators::less<int>{}`.

Sort: RAJA provides in-place sorting of arrays or pairs

```
array = {5, 2, 3A, 1, 3B};
```

Unstable sort:

```
RAJA::sort< exec_pol >( RAJA::make_span(array, N) );
```

```
array : 1, 2, 3B, 3A, 5
```

Original ordering of duplicate keys is not guaranteed in unstable sort.

Stable sort:

```
RAJA::stable_sort< exec_pol >( RAJA::make_span(array, N) );
```

```
array : 1, 2, 3A, 3B, 5
```

RAJA also provides `sort_pairs`, which sorts 2 arrays based on the keys in one of the arrays.

If no 2nd operator argument is given, “less” is the default (non-decreasing order).

More Advanced Capabilities and the RAJA Portability Suite

Shared or stack local memory can be accessed by all threads in a launch


```
launch<launch_policy>(
  Grid(Teams(NTeams), Threads(NThreads)))
  [=] RAJA_HOST_DEVICE(LaunchContext ctx) {

  RAJA_TEAM_SHARED double temp_array[N+1];

  temp_array[0] = 0.0;

  loop<row_policy>(ctx, N_range, [&](int i){
    temp_array[i+1] = myfunc(i+1);
  });

  ctx.teamSync();

  loop<row_policy>(ctx, N_range, [&](int i){
    out_array[i] = temp_array[i+1] – temp_array[i];
  });

});
```

RAJA_TEAM_SHARED

On the CPU, this is a stack local array.

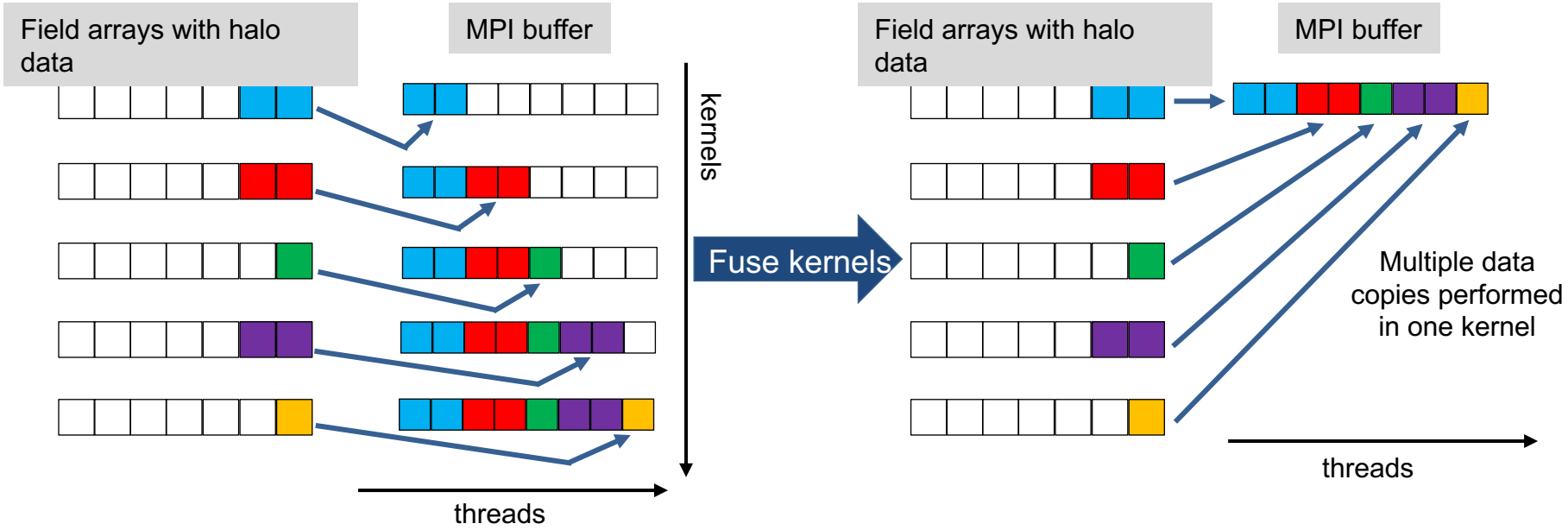
On the GPU, this is a shared memory array which can be accessed by all threads within a Team (block).

Size needs to be known at compile time, but this will change in the next release.

An analogous version exists for RAJA::kernel, i.e. `LocalArray`.

Kernel fusion: Fusing small GPU kernels into one kernel launch helps alleviate negative impact of launch overhead

Key application use case: packing/unpacking halo (ghost) data on a GPU into MPI buffers



One kernel launched per buffer packing operation

One kernel launched for all packing operations

Kernel fusion: RAJA kernel fusion integrates into applications easily

Typical pattern that launched many kernels to pack MPI buffers

```
for ( neighbor : neighbors ) {
  double* buf = buffers[neighbor];
  for ( f : fields[neighbor] ) {
    int len = f.ghostLen();
    double* ghost_data = f.ghostData();
    forall(Range(0, len), [=](int i){
      buf[ i ] = ghost_data[ i ];
    });
    buf += len;
  }
  send(neighbor);
}
```

In WSC production apps, this technique yields 5 - 15% overall run time reduction in typical problems.

Fusing the kernels, runs them in one GPU kernel launch

```
RAJA::WorkPool< ... > fuser;
for ( neighbor : neighbors ) {
  double* buf = buffers[neighbor];
  for ( f : fields[neighbor] ) {
    int len = f.ghostLen();
    double* ghost_data = f.ghostData();
    fuser.enqueue(Range(0, len), [=](int i){
      buf[ i ] = ghost_data[ i ];
    });
    buf += len;
  }
}
auto workgroup = fuser.instantiate();
workgroup.run();
for ( neighbor : neighbors ) {
  send(neighbor);
}
```

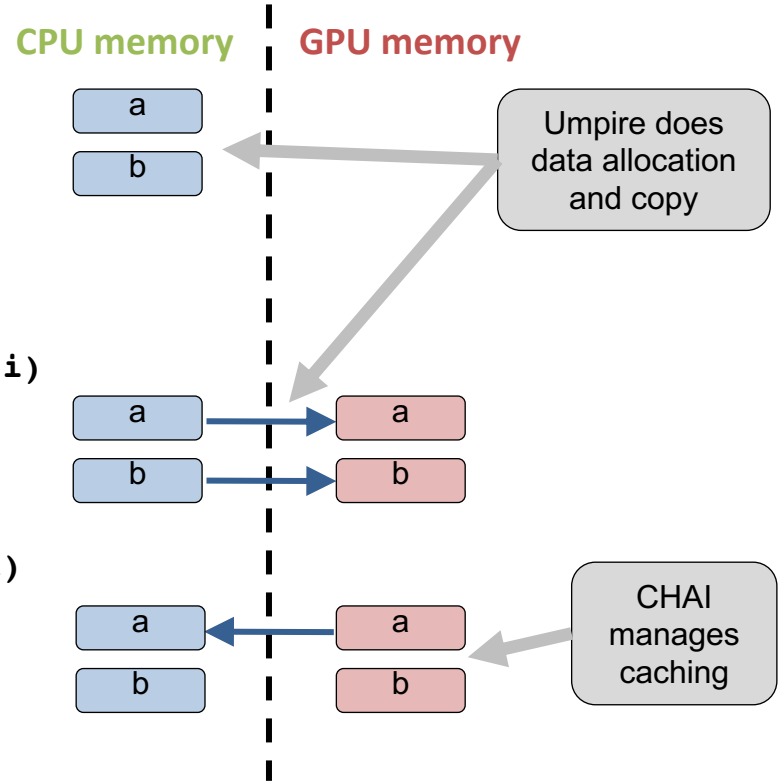
Umpire interface concepts allow application developers to reason about memory use

- A **Memory Resource** is a kind of memory, with specific performance and accessibility characteristics
- An **Allocation Strategy** decouples how and where allocations are made, allowing complex allocation mechanisms
- An **Allocator** is a lightweight interface for making an allocation and querying it
  - One interface for all resources
- An **Operation** manipulates data in memory through one interface regardless of resource
  - Copy, move, reallocate, memset, etc.
- These concepts are coordinated by a **ResourceManager**
  - Builds allocators based on allocation strategies and available resources, dispatches operations based on pointer locations, etc.

```
auto& rm = umpire::ResourceManager::getInstance();
auto host = rm.getAllocator("HOST");
auto device = rm.getAllocator("DEVICE");

auto device_pool =
  rm.makeAllocator<QuickPool>("MY_POOL", device);


void* host_data = host.allocate(1024);
void* dev_data = device_pool.allocate(1024);


rm.memset(host_data, 0);
rm.copy(dev_data, host_data);


host.deallocate(host_data);
```

CHAI's "managed array" abstraction copies data automatically as needed to run kernels

```
chai::ManagedArray<float> a(100);
chai::ManagedArray<const float> b(100);

// ...
RAJA::TypedRangeSegment<int> range(0, 100);

// Run GPU kernel
RAJA::forall<RAJA::cuda_exec>(range,
                              RAJA_LAMBDA (int i)
  { a[i] += b[i]; }
);

// Run CPU kernel
RAJA::forall<RAJA::seq_exec>(range,
                             RAJA_LAMBDA (int i)
  { std::cout << "a , b = " << a[i] << " , "
                            << b[i] << "\n"; }
);
```



CHAI's "managed pointer" simplifies the use of virtual class hierarchies across host and device memory spaces

- `managed_ptr` will make a copy of object hierarchy in device memory

```
void overlay( Shape* shape, double* mesh_data ) {
  chai::managed_ptr< Shape > mgd_shape = shape->makeManaged();
  RAJA::forall< cuda_exec > ( ...  {
    mgd_shape->processData(mesh_data[i]);
  } );
  mgd_shape.free();
}
```

- This requires a method to clone objects and adding host-device annotations to class constructors

```
chai::managed_ptr< Shape > Sphere::makeManaged( ) { ... }

__host__ __device__ Sphere::Sphere( ... ) { ... }
```

This mechanism allows the use of C++ virtual class hierarchy code on both CPUs and GPUs without a major refactor.

RAJA asynchronous execution allows users to control overlap of kernel execution and memory transfers (CHAI and Umpire)

Async Exec

```
chai::ManagedArray<double> a1(N);
chai::ManagedArray<double> a2(N);

RAJA::resource::Cuda cuda1;
RAJA::resource::Cuda cuda2;

auto event1 = forall<cuda_exec_async>(&cuda1, RangeSegment(0, N),
                                      [=] RAJA_DEVICE (int i) { a1[i] = ... } );

auto event2 = forall<cuda_exec_async>(&cuda2, RangeSegment(0, N),
                                      [=] RAJA_DEVICE (int i) { a2[i] = ... } );

cuda1.wait_on(&event2);

forall<cuda_exec_async>(&cuda1, RangeSegment(0, N),
                        [=] RAJA_DEVICE (int i) { a1[i] *= a2[i]; } );

forall<seq_exec>(RangeSegment(0, N),
                 [=] (int i) { printf("a1[%d] = %f \n", i, a1[i]); } );
```

Resources assigned to different CUDA streams passed to RAJA kernel execution methods

Kernel execution methods return events that one can query or wait on to control execution and synchronization

Application considerations

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

Apps

- 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

Apps

- 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

Apps

- 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?

RAJA promotes flexibility and tuning via type parameterization

Apps

- 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 can be challenging
  - 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 (e.g., GPU execution)
  - 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.

Wrap-up

Materials that supplement this tutorial are available

- Complete working example codes are available in the RAJA source repository
  - https://github.com/LLNL/RAJA
  - Many similar to the examples we presented today
  - 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)

Other related software that may be of interest

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

The RAJA Performance Suite is a good source of examples for many RAJA usage patterns.

Again, we would appreciate your feedback…

- If you have comments, questions, suggestions, etc., please talk to us
- The best way to contact us is via our team email list: raja-dev@llnl.gov

Thank you for your attention and participation

Questions?