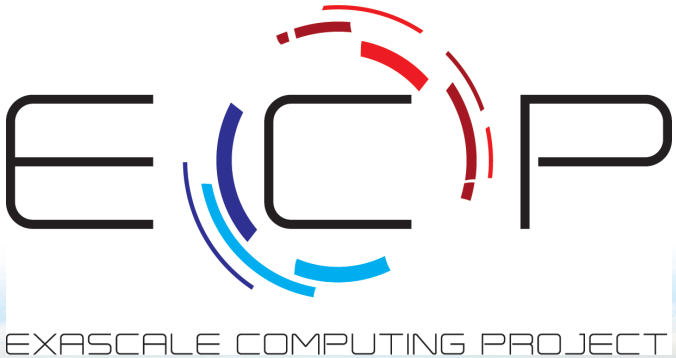
A Tutorial Introduction to RAJA

ATPESC

August 2, 2018



Rich Hornung (hornung1@llnl.gov)
David Beckingsale(beckingsale1@llnl.gov)
With contributions from others on the RAJA Team

LLNL-PRES- 744710
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

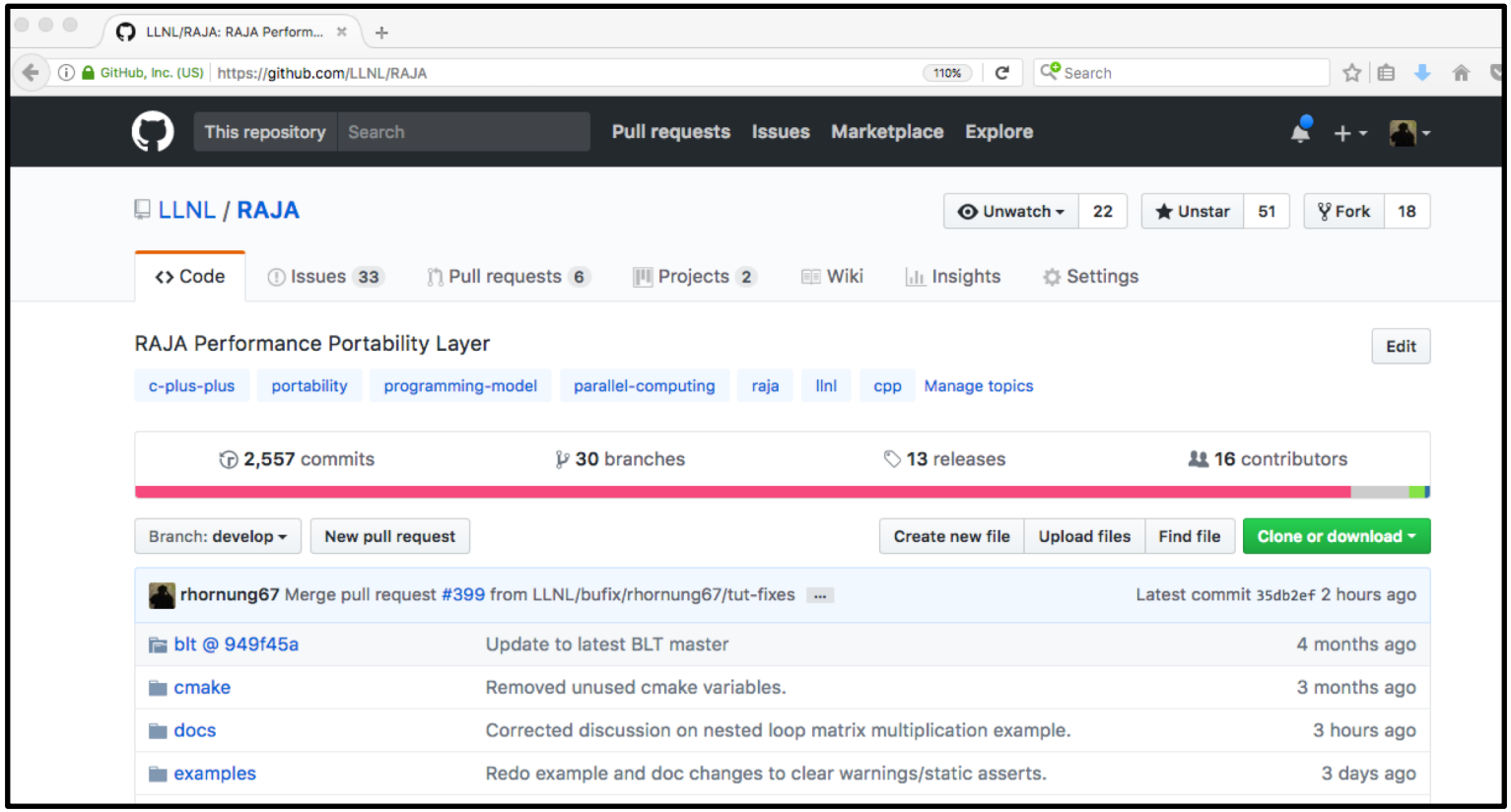
Welcome to the RAJA tutorial

- Today, you will learn how to use key RAJA features:
  - Parallel loop execution
    - Simple loops (e.g., non-nested)
    - Complex loop kernels (composition and transformation)
  - Reductions
  - Atomic operations
  - Scan operations
- We will also briefly discuss:
  - RAJA motivation and goals
  - Application considerations for C++ portability solutions like RAJA

See the RAJA User Guide to learn about features not covered today (https://readthedocs.org/projects/raja).

RAJA is an open source project hosted on Github

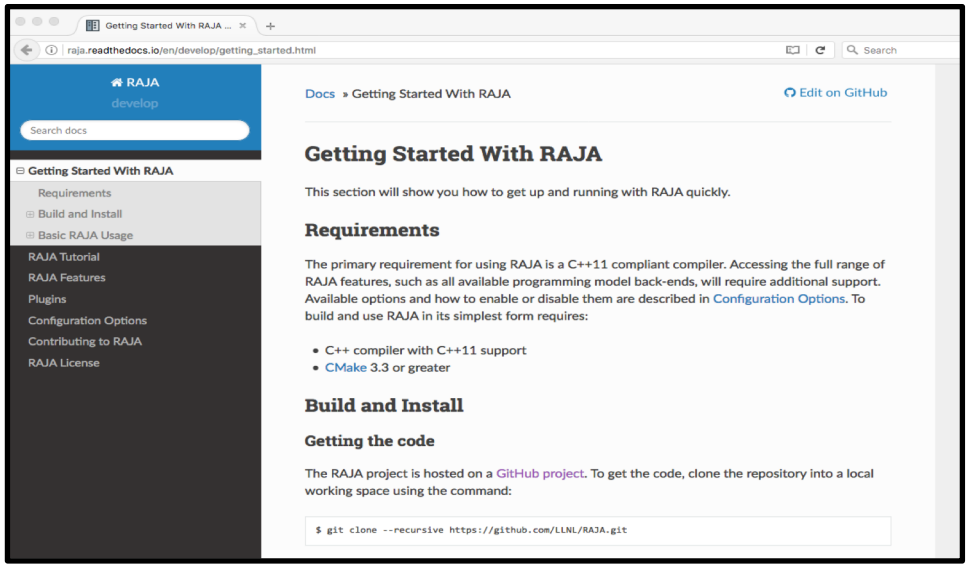
- The “RAJA/examples” directory contains codes used in this tutorial



https://github.com/LLNL/RAJA

Other materials and related projects are also available…

- **RAJA User Guide:** getting started info and details about today's topics and more
- **RAJA Performance Suite:** Monitor RAJA performance and assess compilers, used by vendors, used in DOE platform procurements



- **RAJA Proxy Apps:** RAJA versions of DOE proxy apps
- **CHAI:** Array abstraction library that automatically migrates data as needed based on RAJA execution contexts

These are linked on RAJA Github project.

We value your feedback…

- If you have comments, questions, suggestions, etc., please let us know
  - Join our Google Group (linked on RAJA Github project)
  - Or contact us using our project email list: raja-dev@llnl.gov

During today's presentation…

Please don't hesitate to ask questions at any time

Let's start simple…

Simple loop traversal

RAJA encapsulates loop execution details using standard C++ features

Daxpy operation: a = a + c * b; a, b are vectors of length N, c is a scalar

C-style for-loop

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

The way the loop is expressed is different, but the loop body is the same.

RAJA-syle loop

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

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

There are four core elements to RAJA loop execution

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

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

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

Elements of RAJA loop execution

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

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

Elements of RAJA loop execution

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

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

Elements of RAJA loop execution

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

The programmer must make sure the loop body works with the execution policy.

- RAJA::forall template runs loop iterations 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**
  - A *closure* that stores a function with a data environment
  - Function argument is the loop variable

These core elements will be common threads in our discussion

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

- **RAJA::forall** template runs loop iterations 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**

By changing the execution policy, a loop can run in different ways

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

- Some execution policy choices…

```
RAJA::simd_exec
```

```
RAJA::omp_parallel_for_exec
```

```
RAJA::cuda_exec<BLOCKSIZE>
```

```
RAJA::omp_target_parallel_for_exec<NUMTEAMS>
```

```
RAJA::tbb_for_exec
```

RAJA provides a variety of execution policy types for simple loops...

- Sequential (strictly)
- "Loop" (let compiler decide which optimizations to apply)
- SIMD (vectorization pragmas applied)
- OpenMP multithreading (CPU)
- TBB (Intel Threading Building Blocks)
- CUDA
- OpenMP target (available target device; e.g., GPU)

- Some are works-in-progress:
  - OpenMP target (work with IBM)
  - TBB (work with Intel)
  - ROCm will be available in a future release (work with AMD)

RAJA support for simple loops

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

= available = in progress = not available

RAJA
Motivation and Goals

RAJA targets portable, parallel loop execution

- Overarching goals of RAJA:
  - —***Enable portability*** in existing applications with ***manageable disruption*** to algorithms and programming styles
  - —Provide a model for new applications to be ***portable from the start***

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

RAJA design points focus on usability and developer productivity

- Applications should maintain **single-source kernels** (if possible)
- **Easy to grasp** for app developers (most are not CS experts)
- Allow for **incremental and selective** use
- **Don't force major disruption** to application source code
- Promote implementation flexibility via **clean encapsulation**
- Make it **easy to parameterize execution** via types
- Enable **systematic performance tuning**

Our focus on these points has enabled RAJA adoption.

Reductions

RAJA reduction types hide the complexity of parallel reduction implementations

Dot product: dot = (a, b), where a and b are vectors and dot is a scalar

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 >( arange, [=] (int i) {
  dot += a[i] * b[i];
} );
```

Elements of RAJA reductions…

```
RAJA::ReduceSum< reduce_policy, type > sum(init_val);

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

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

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

Elements of RAJA reductions…

```
RAJA::ReduceSum< reduce_policy, type > foo(init_val);

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

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

- Each 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, type > sum(init_val);

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

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

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

Elements of RAJA reductions…

```
RAJA::ReduceSum< reduce_policy, type > sum(init_val);

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

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

Reduction policy **must be compatible** with loop execution policy.

RAJA provides reduction policies for all programming model back-ends

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

- Reduction policies

```
RAJA::seq_reduce;
```

```
RAJA::omp_reduce;
```

```
RAJA::cuda_reduce<BLOCK_SIZE>;
```

```
RAJA::tbb_reduce;
```

```
RAJA::omp_target_reduce<NUMTEAMS>;
```

Note: SIMD, OpenMP target, and ROCm are works-in-progress.

RAJA supports five common reductions types

```
RAJA::ReduceSum<    reduce_policy, type > r(in_val);

RAJA::ReduceMin<    reduce_policy, type > r(in_val);

RAJA::ReduceMax<    reduce_policy, type > r(in_val);

RAJA::ReduceMinLoc< reduce_policy, type > r(in_val,
                                            in_loc);

RAJA::ReduceMaxLoc< reduce_policy, type > r(in_val,
                                            in_loc);
```

Initial “loc” values

“Loc” reductions **give iteration index** where reduced value was found.

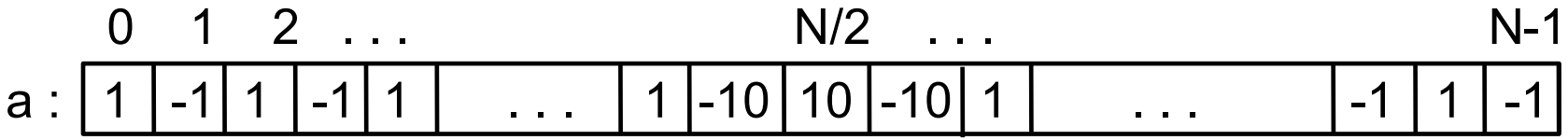
Multiple RAJA reductions can be used in a kernel

```
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 >( a_range, [=](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);
} );
```

Remember: Reduction and loop execution policies must be compatible.

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

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



- This will yield
  - Sum = -9
  - Min = -10
  - Max = 10
  - Max-loc = N / 2
  - Min-loc = N / 2 - 1 (or N / 2 + 1)*

RAJA provides **reproducible parallel reductions** if you need them.

RAJA support for reductions

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

= available = in progress = not available

Iteration spaces : Segments and IndexSets

A RAJA "Segment" is the basic means to define a loop iteration space

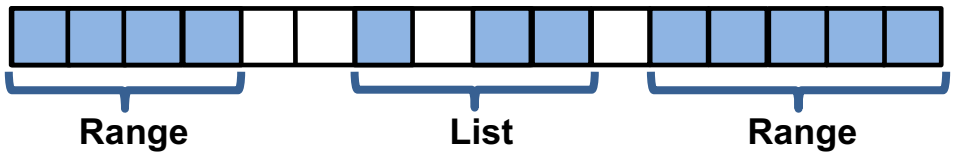
- A "Segment" defines a set of loop indices to run as a unit

Contiguous range [beg, end)

Strided range [beg, end, stride)

List of indices (indirection)

- An "Index Set" is a container of segments



You can run all segments in an IndexSet in one RAJA traversal.

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

```
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

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

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

Runs loop indices: {0, 2, 4, …}

```
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}

RAJA supports negative indices and strides.

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 defines a set of indices (think "indirection array")

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

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

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

A RAJA Index Set is a container of Segments

```
RAJA::RangeSegment range1(0, 8);

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

RAJA::RangeSegment range3(24, 28);
```

```
RAJA::TypedIndexSet< RAJA::RangeSegment,
                     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

An IndexSet can be passed to a RAJA traversal method to run all Segments

```
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**:

- One for iterating over segments ("…_segit")
- One for executing segments

RAJA Segment types model C++ iterable interfaces

- A segment type defines three methods:
  - begin()
  - end()
  - size()
- And two types:
  - iterator (essentially “random access”)
  - value_type

Many **user-defined types** that have these properties can be used as a Segment with RAJA.

Why do we provide Index Sets?

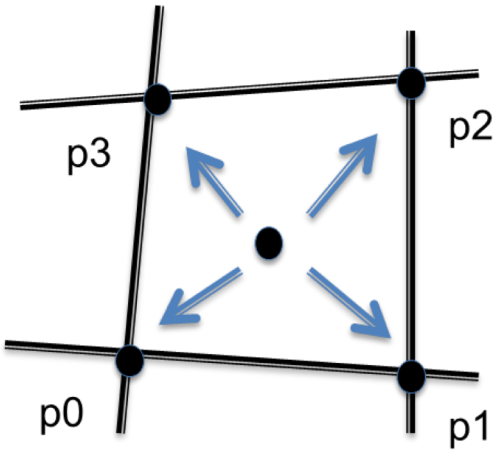
- **Multiphysics codes use indirection arrays** (a lot!) : simplicity & generality – unstructured meshes, material regions on a mesh, etc.
  - Indirection impedes performance: extra instructions, no SIMD, etc.
- **Range Segments are better for performance**
  - When large stride-1 ranges are embedded in iteration patterns…
  - …you can expose 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) – contention heavy
  - Avoid extra arrays and gather/scatter operations – extra memory traffic
  - Prefer coarse-grained synchronization - light memory contention

With IndexSets, you can change the iteration pattern **without changing the way it looks** in source code.

IndexSets help enable thread parallelism…

Example: accumulate average element volumes to mesh vertices

```
for (int j = 0 ; j < N_elem ; ++j) {
  for (int i = 0 ; i < N_elem ; ++i) {

    int ie = i + j * jeoff ;
    int* iv = &(elem2vert_map[4*ie]);

    vertvol[ iv[0] ] += elemvol[ie] / 4.0 ;
    vertvol[ iv[1] ] += elemvol[ie] / 4.0 ;
    vertvol[ iv[2] ] += elemvol[ie] / 4.0 ;
    vertvol[ iv[3] ] += elemvol[ie] / 4.0 ;
  }
}
```

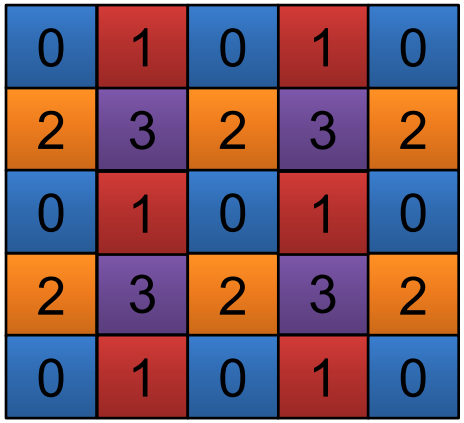


As written, this code **will not run correctly** in parallel – **data races!**

Partition elements into independent subsets (colors) to avoid race conditions

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

RAJA::forall<EXEC_POL>(colorset, [=](int ie) {

  int* iv = &(elem2vert_map[4*ie]);
  vertexvol[ iv[0] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[1] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[2] ] += elemvol[ie] / 4.0 ;
  vertexvol[ iv[3] ] += elemvol[ie] / 4.0 ;

} );
```

Important: the loop body looks like the domain expert wrote it!



RAJA Segments and IndexSets work with all back-ends

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

= available = in progress = not available

Atomic operations

RAJA provides portable atomic operations

```
int* x = ...
int* y = ...

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

  RAJA::atomic::atomicAdd< atomic_policy >(&x[i], 1);
  RAJA::atomic::atomicSub< atomic_policy >(&y[i], 1);

} );
```

Atomic operations perform atomic memory updates (write or read-modify-write).

Atomics may be preferable to reductions

Example: compute a histogram of array values

```
int* a    = ...  // array length N (a[i] in {0,..,M-1})
int* bins = ...  // array length M (M <= N)

using EXEC_POL   = RAJA::cuda_exec;
using ATOMIC_POL = RAJA::atomic::cuda_atomic;

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

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

} );
```

This is simpler (and may perform better) than using reductions.

The atomic policy determines the atomic implementation

- 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) {
  atomic::atomicAdd< RAJA::builtin_atomic >(sum, 1);
} );
```

The atomic policy must be compatible with loop execution policy.

RAJA also has an "auto" atomic policy

- The RAJA::auto_atomic policy will do the right thing…

```
using EXEC_POL = RAJA::cuda_exec;

int *sum = ...;

RAJA::forall< EXEC_POL >(arange, [=] RAJA_DEVICE(int i) {
  atomic::atomicAdd< RAJA::auto_atomic >(sum, 1);
} );
```

Some may prefer this option for easier portability.

RAJA also has an interface modeled after C++ std::atomic

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

For example:

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

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

Result: sum is 5.

RAJA provides a variety of atomic operations

- Arithmetic: add, sub
- Min, max
- Increment/decrement: inc, dec, including comparisons
- Bitwise-logical: and, or, xor
- Replace: exchange, CAS
- C++ std::atomic style interface

RAJA User Guide describes these atomic operations in detail.

RAJA support for atomics

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

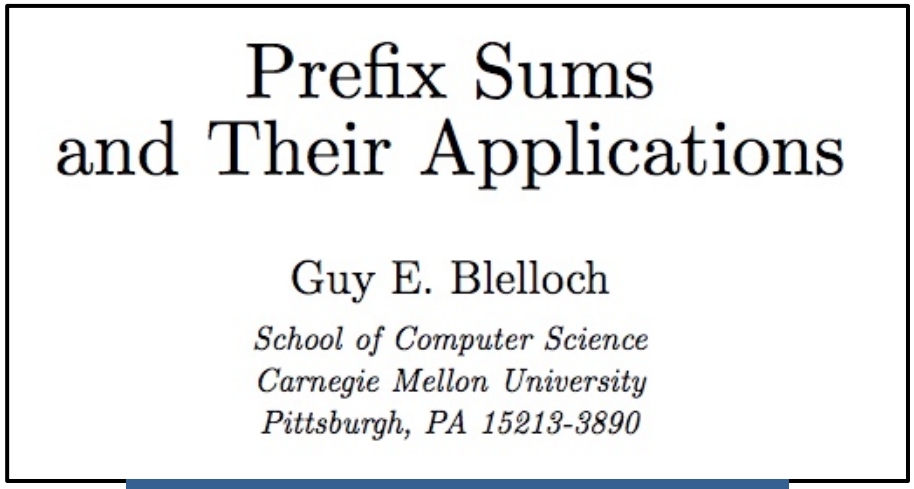
= available = in progress = not available

Scan operations

Scan is a basic parallel algorithm building block

- Key primitive to convert serial operations to parallel
  - Based on reduction tree and reverse reduction tree

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


Prefix Sums
and Their Applications

Guy E. Blelloch

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Blelloch's Lecture Notes are worth reading.

Parallel prefix sum is the most common scan

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

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

Output array contains partial sums of input array.

RAJA provides "in-place" scan operations

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

RAJA::inclusive_scan_inplace< exec_pol >(in, in + N);

RAJA::exclusive_scan_inplace< exec_pol >(in, in + N);
```

"In-place" scans return **result in input array**.

RAJA scans work with different operators

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

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

Out : 8 -1 -1 -1 -1 -1 -1 -1 -1 -1

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

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

Out : -2147483648 8 8 8 9 10 10 10 10 10

If no operator is given, **"plus" is the default (prefix-sum).**

RAJA support for scans

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

= available = in progress = not available

Complex Loops

Let's look at a nested loop example and apply what we've seen so far…

Matrix multiplication: C = A * B, where A, B, C are N x N matrices

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

  }
}
```

We could nest 'forall' statements…

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

  } );
} );
```

…but, this doesn't work well

- ***Each loop level is treated as an independent entity***
  - So parallelizing the row and column loops together is hard
- We can use a parallel execution policy (OpenMP, CUDA, etc.) on the outer row loop
  - Then, each thread executes all code inside it **sequentially**
- Parallelizing the inner column loop introduces potential synchronization for each outer loop iteration
  - Launch a separate parallel computation for each row
- Loop interchange and other transformations **require changing the source code of the kernel (which breaks RAJA encapsulation)**

We don't recommend using RAJA::forall for nested loops!!

Complex loops

The RAJA::kernel API is used to compose and transform complex parallel kernels

```
using KERNEL_POL =
    RAJA::KernelPolicy<
      RAJA::statement::For<1, exec_policy_row>,
        RAJA::statement::For<0, exec_policy_col>,
          RAJA::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[k + N*row] * B[col + N*k];
    }
    C[col + N*row] = dot;
} );
```

Note: lambda expression for inner loop body is same as before.

There are four key elements to the RAJA::kernel interface

- These are analogous to RAJA::forall

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

Each loop level has an iteration space, and loop variable

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

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

Order (and types) of tuple items and loop variables must match.

Each loop level has an execution policy…

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

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

'0' → col
'1' → row

Integer parameter in 'For' statement specifies the tuple item it applies to.

Loop interchange done by changing the execution policy, not algorithm code

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

'For' statements are swapped.

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

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

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

A kernel can be composed from any number of lambdas

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

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

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

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

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

All lambdas must have the **same argument list**.

The execution policy composes 'statements' into a kernel

```
using KERNEL_POL =
  RAJA::KernelPolicy<
    RAJA::statement::For<1, exec_policy_row,
      RAJA::statement::For<0, exec_policy_col,

        RAJA::statement::Lambda<0>,  // init: dot = 0.0
        RAJA::statement::For<2, RAJA::seq_exec,
          RAJA::statement::Lambda<1> // inner loop: dot += ...
        >,
        RAJA::statement::Lambda<2>   // C[col + N*row] = dot;

      >
    >
  >;
```

Nesting 'RAJA::statement' types is analogous to nesting for-loops and other statements in a C-style loop nest.

You can collapse loops in an OpenMP parallel region…

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

      RAJA::statement::Lambda<0>,  // dot = 0.0
      RAJA::statement::For<2, RAJA::seq_exec,
        RAJA::statement::Lambda<1> // inner loop: dot += ...
      >,
      RAJA::statement::Lambda<2>  // set C[col + N*row] = dot;

    >
  >;

RAJA::kernel_param<KERNEL_POL>(
  RAJA::make_tuple(col_range, row_range, dot_range),
  // lambdas used to compose kernel body...
);
```

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

You can launch loops as a CUDA kernel…

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

This policy distributes ‘row’ indices over CUDA thread blocks and ‘col’ indices over threads in each block; i.e., same as defining indices inside the kernel as:

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

Another CUDA kernel variant…

```
using KERNEL_POL =
  RAJA::KernelPolicy<
    RAJA::statement::CudaKernel<
      RAJA::statement::For<1, RAJA::cuda_threadblock_exec<BS>,
        RAJA::statement::For<0, RAJA::cuda_threadblock_exec<BS>,
          RAJA::statement::Lambda<0>,    // dot = 0.0
          RAJA::statement::For<2, RAJA::seq_exec,
            RAJA::statement::Lambda<1>  // dot += ...
          >,
          RAJA::statement::Lambda<2>     // set C = ...
        >
      >
    >
  >;
```

This policy distributes ‘row’ and ‘col’ indices over 2-dimensional BS X BS CUDA thread blocks.

RAJA kernel policies combine 'Statements' and 'StatementLists'

- A **Statement is an action**: execute a loop, invoke a lambda, set a thread barrier, etc. For example,

```
Lambda<0>
```

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

```
For<0, exec_policy0,
   Lambda<0>,
   For<2, exec_polic2,
      Lambda<1>
   >
>
```

- These policy constructs form a simple domain specific language (DSL) that relies only on standard C++11 support

A RAJA::KernelPolicy type **is a StatementList**.

The current list of RAJA::statement types...

- **For< Id, ExecPolicy, EnclStmts >** – abstracts a for-loop
- **Lambda< Id >** – invoke a lambda expression
- **Collapse< ExecPolicy, ArgList<…>, EnclStmts >** – collapse multiple, perfectly nested loops
- **If< Conditional >** – select parts of a policy to use at runtime
- **CudaKernel< EnclStmts >** – launch a CUDA kernel
- **CudaSyncThreads** – CUDA sync threads barrier (similar for OpenMP will be added)

The current list of RAJA::statement types...

- **SetShmemWindow< EnclStmts >** – set window into a shared memory buffer
- **Tile< Id, TilePolicy, ExecPolicy, EnclStmts >** – tile (or cache blocking) of an outer loop
- **Hyperplane< Id, HpExecPolicy, ArgList<…>, ExecPolicy, EnclStmts >** – hyperplane iteration over multiple indices

Other statement types will appear in RAJA as new use cases arise.

RAJA support for complex loops

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

= available = in progress = not available

Views and Layouts

Matrices and tensors are ubiquitous in scientific computing applications

- They are expressed most naturally as multi-dimensional arrays
- For efficiency in C/C++, they are usually allocated as 1-d arrays.

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

Recall matrix multiplication example

```
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;
```

- Here, we manually convert 2-d indices (row-col) to a pointer offset
- We could also use a macro… `#define A(r, c) A[c + N*r]`
- Or…

RAJA Views and Layouts enable a variety of multi-dimensional indexing patterns

- A RAJA View enables multi-dimensional indexing into an array based on a RAJA Layout type

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

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

- This leads to a natural indexing method

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

RAJA Views and Layouts support any number of dimensions

```
double* A = new double[ N0 * ... * Nn ];

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

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

// 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)

The right-most index is stride-1 by default.

RAJA provides methods to make layouts for other indexing patterns

- **Permuted layout**

```
RAJA::Layout< 3 > perm_layout =
  RAJA::make_permuted_layout(
    {{5, 7, 11}},
    RAJA::as_array< RAJA::Perm<1, 2, 0> >::get() );
```

A 3-d layout with indices permuted:

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

The “identity” permutation (e.g., RAJA::Perm<0, 1, 2>) gives a layout the same as if no permutation is used.

A RAJA permuted layout example…

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

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

RAJA::Layout<3> perm_layout =
  RAJA::make_permuted_layout( {{s0, s1, s2}},
    RAJA::as_array< RAJA::Perm<1, 2, 0> >::get() );

RAJA::View< double,
            RAJA::Layout<3, int, 0> > Bview(B,
                                            perm_layout);
// Equivalent to indexing as: B[i + j*s0*s2 + k*s0]
Bview(i, j, k) = ...;
```

Args are dimension, index type, and stride-1 index (optional optimization)

A RAJA offset layout applies offsets to indices

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

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

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

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

A 1-d layout with extent 11 and index offset.
-5 is subtracted from each loop index to access data.

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

RAJA offset layouts support permutations too

```
RAJA::Layout<2> offset_layout2 =
  RAJA::make_offset_layout<2>( {{-1, -5}}, {{2, 5}} );
```

A layout for 2-d index space [-1, 2] X [-5, 5].
Index '1' **(right-most)** has stride-1 and index '0' has stride 11.

```
RAJA::Layout<2> permoffset_layout2 =
  RAJA::make_offset_layout<2>( {{-1, -5}}, {{2, 5}});
    RAJA::as_array< RAJA::Perm<1, 0> >::get() );
```

A layout for 2-d index space [-1, 2] X [-5, 5] (same as above.
But, now index '0' has stride-1 and index '1' has stride 4.

As before, when no permutation is given, the convention is that the right-most View index has stride-1.

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

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

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

```
// Convert i=2, j=3, k=1 to 1-dim index:
//   lin = 188 (= 1 + 3 * 11 + 2 * 11 * 7)
int lin = layout(2, 3, 1);


// Convert linear index to 3d indices:
//   i, j, k = {2, 3, 1}
layout.toIndices(lin, i, j, k);
```

RAJA layout also support "projections"

- When a dimension has **zero extent**, the linear index space is **invariant in that dimension**

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

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

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

Application considerations

RAJA is motivated by constraints of large, production, multi-physics applications

Apps

- **Large code bases**
  - O(100K) – O(1M) SLOC and many numerical kernels (O10K)
- **Platform diversity**
  - Codes run on laptops, commodity clusters, advanced technology systems
  - ASC program procures large commodity and advanced technology systems in 3-5 year cycles
- **Long service lives**
  - Codes used in production daily for decades – must be viable over several platform generations
- **Continual development**
  - New modeling capabilities added to codes throughout their lifetimes
  - Adopting new technologies (h/w & s/w) cannot disrupt users

RAJA design emphasizes usability and developer productivity

- **Single-source kernels** (no platform-specific variants if possible)
- **Easy to grasp** for (non-CS) application developers
- Allows for **incremental and selective** adoption
- Provides necessary **features not found in other models**
- **Doesn't force major disruption** to application algorithm patterns or data structures
  - Loop bodies are unchanged in most cases
  - Works with existing code constructs
  - Allows application-specific customizations

Application developers typically wrap RAJA in a layer to **match their code's style**.

RAJA promotes flexibility via type parameterization

- Define **type aliases in header files**
  - Easy to explore implementation choices in a large code base
  - Reduce 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 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 appropriate "loop taxonomy" and policy selection for their code.

"Bring your own" memory management

- RAJA does not provide a memory model (by design)
  - Users must manage memory space allocations and transfers

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

Are 'a' and 'b' accesible on GPU?

"Bring your own" memory management

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

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

Are 'a' and 'b' accesible on GPU?

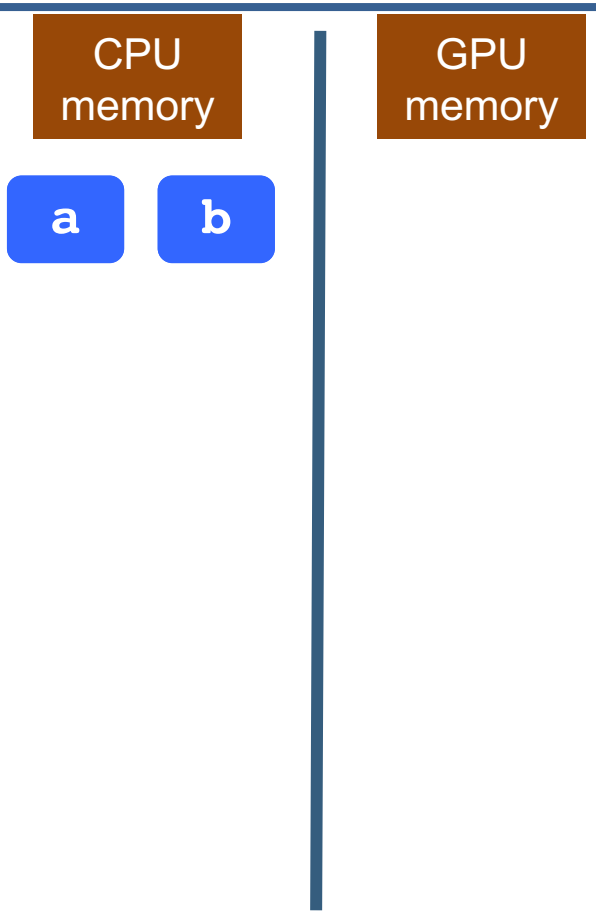
- Some possibilities:
  - **Manual** – use cudaMalloc( ), cudaMemcpy( ) to allocate, copy to/from device
  - **Unified Memory (UM)** – use cudaMallocManaged( ), paging on demand
  - **CHAI** (github.com/llnl/chai) – automatic copies as needed

CHAI was developed to complement to RAJA.

CHAI provides array abstractions for transparent, automatic data copies

Apps

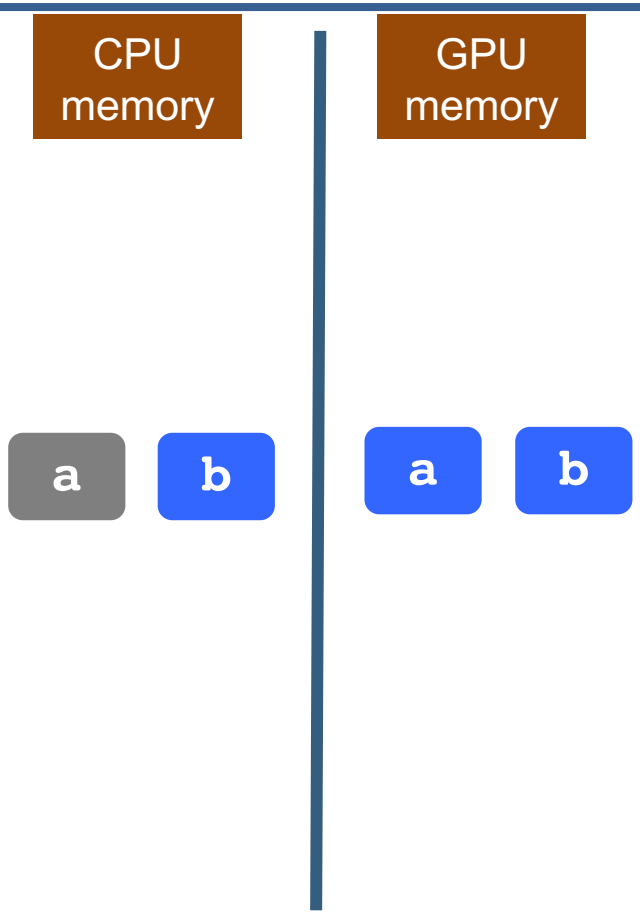
```
chai::ManagedArray<int> a...;
chai::ManagedArray<const int> b...;

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

RAJA::forall<RAJA::seq>(range,
  [=] (int i) {
    printf("%d, %d \n", a[i],b[i]);
} );
```



CHAI provides array abstractions for transparent, automatic data copies

Apps

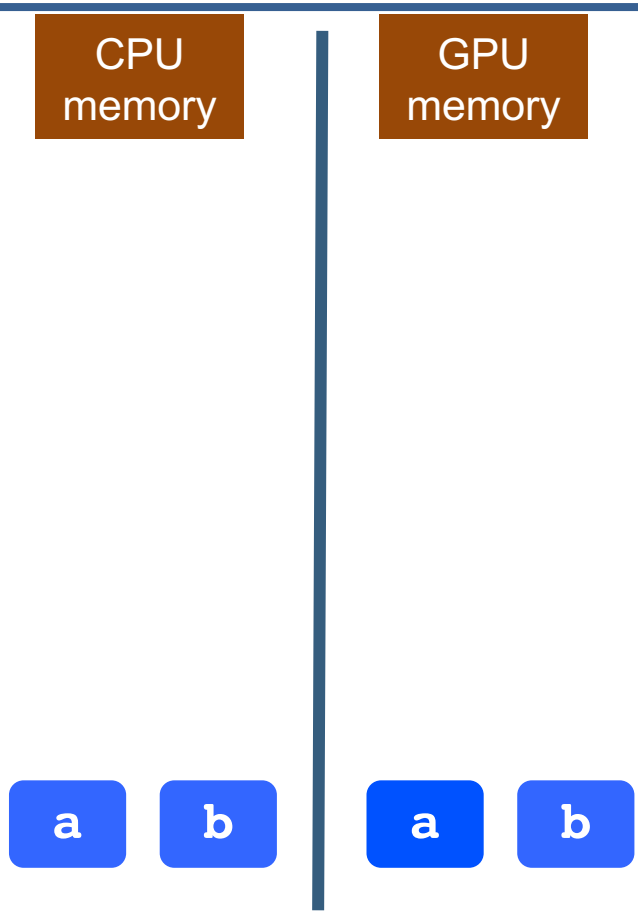
```
chai::ManagedArray<int> a...;
chai::ManagedArray<const int> b...;

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

RAJA::forall<RAJA::seq>(range,
  [=] (int i) {
    printf("%d, %d \n", a[i],b[i]);
} );
```



CHAI provides array abstractions for transparent, automatic data copies

Apps

```
chai::ManagedArray<int> a...;
chai::ManagedArray<const int> b...;

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

RAJA::forall<RAJA::seq>(range,
  [=] (int i) {
    printf("%d, %d \n", a[i],b[i]);
} );
```



CHAI supports UM too, so you can assess its performance.

C++ lambda usage requires care

- Capture by-value or by-reference (**[=] vs. [&]**)?
  - **Value capture is required** for GPU execution and when using RAJA reductions, when using CHAI, …

C++ lambda usage requires care

- Capture by-value or by-reference (**[=] vs. [&]**)?
  - **Value capture is required** for GPU execution, when using RAJA reductions, when using CHAI, …
- You **cannot** use ‘break’ or ‘continue’ statements in a lambda

C++ lambda usage requires care

- Issue: global variables aren't captured (C++ standard):

```
extern double global_var;

RAJA::forall<RAJA::cuda_exec>(range, [=] (int i) {
  a[i] += global_var;
} );
```

- Solution: make a local reference:

```
double& ref_to_global_var = global_var;

RAJA::forall<RAJA::cuda_exec>(range, [=] (int i) {
  a[i] += ref_to_global_var;
} );
```

For CPU code, you get lucky (maybe).

C++ lambda usage requires care

- Issue: local stack arrays aren't captured for CUDA device code (nvcc compilation error):

```
int[4] bounds = { 0, 1, 8, 9};

RAJA::forall<RAJA::cuda_exec>(range, [=] __device__ (int i) {
  for (int bound = 0; bound < 4; bound++) {
    my_bound = bounds[bound];
  }
});
```

C++ lambda usage requires care

- Solution: wrap the array in a struct:

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

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

RAJA::forall<RAJA::cuda_exec>(range, [=] __device__ (int i) {
  for (int bound = 0; bound < 4; bound++) {
    my_bound = bounds.array[bound];
  }
});
```

CUDA device lambdas need annotations

- A lambda passed to a CUDA device function **must have the "__device__"** decoration
  - e.g., when using RAJA CUDA execution policies
- A lambda can also be marked **"__host__ __device__"**
- **Beware of using a "__host__ __device__" lambda in host code**
  - The CPU code will be much slower than if you use an undecorated lambda

RAJA provides macros to help with this.

Performance portability takes effort

- RAJA (like other similar approaches) is **an enabling technology – not a panacea**
  - Loop characterization and performance tuning are manual processes
    - Good tools are essential…
  - Memory motion is critical. Pay attention to it!

Performance portability takes effort

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

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

Wrap-up

RAJA features are supported for a variety of programming model back-ends

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	ROCm
Simple loops							
Reductions							
Segments & Index sets							
Atomics							
Scans							
Complex Loops							
Layouts & Views							

= available = in progress = not available

Supplemental materials to this tutorial are available online

- Complete working example codes are available in the RAJA source repository
  - https://github.com/LLNL/RAJA
  - Many similar to the examples we presented today
  - After cloning the repo (instructions on GitHub), look in the "RAJA/examples" directory
- The RAJA User Guide provides more details
  - Topics we discussed today, configuring & building RAJA, etc.
  - Available online: http://raja.readthedocs.org/projects/raja (also linked on the RAJA GitHub project)

Related software is available

- The RAJA Proxy App Suite
  - RAJA versions of some important HPC proxies (more in the works)
  - https://github.com/LLNL/RAJAProxies
- 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

Related software is available

- 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 at the hands-on session
- You are welcome to join our Google Group: https://groups.google.com/forum/#!forum/raja-users
- Or contact us via email: raja-dev@llnl.gov

Thank you for your attention and participation

Questions?