A Tutorial Introduction to RAJA

RAJA

EXASCALE COMPUTING PROJECT

October, 2023

Presented by Robert Chen
on behalf of the RAJA Team

LLNL-PRES-855469

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 present examples that show you how to use various RAJA features
- Our objective is that you will learn enough about RAJA to start using it in your own code development

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

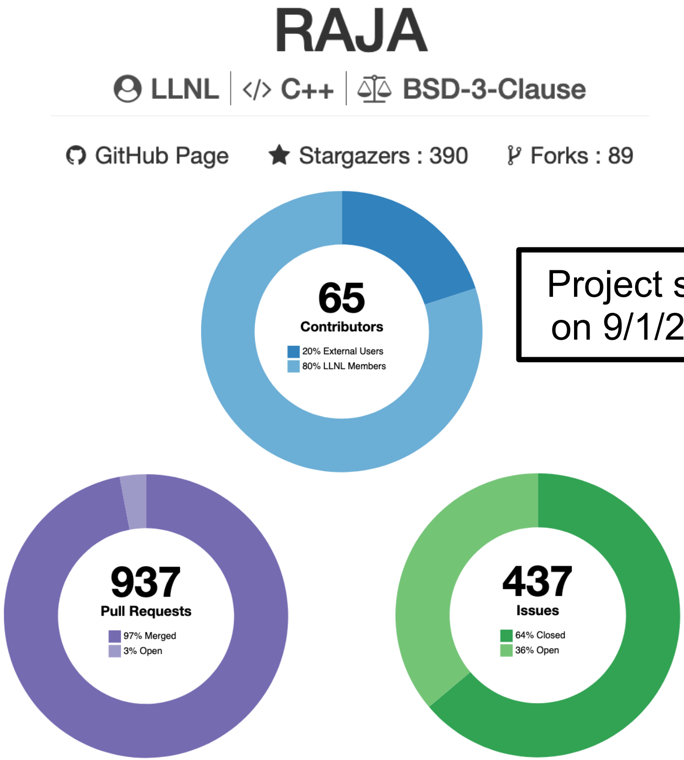
During the tutorial…

Please don't hesitate to ask questions at any time

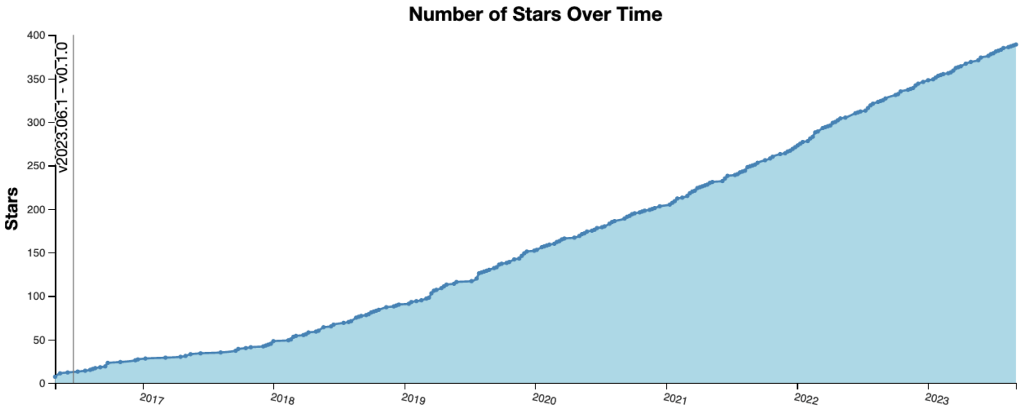
We value your feedback…

- 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 our tutorial materials

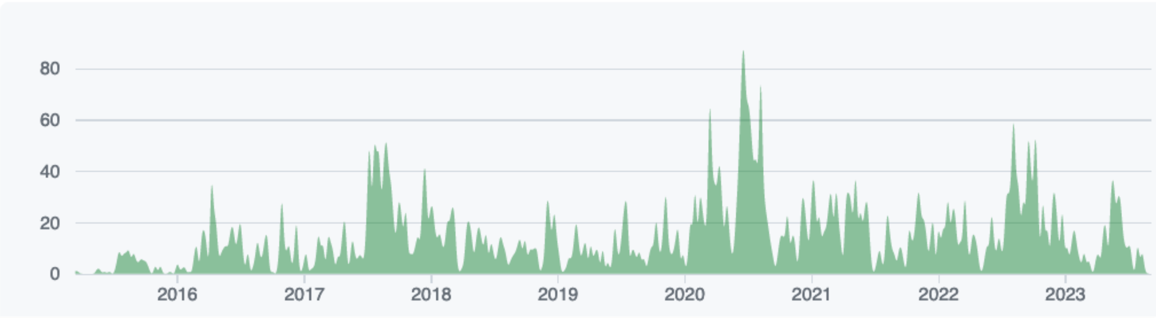
RAJA is an open-source project with a growing user base and set of contributors



Project stats on 9/1/2023

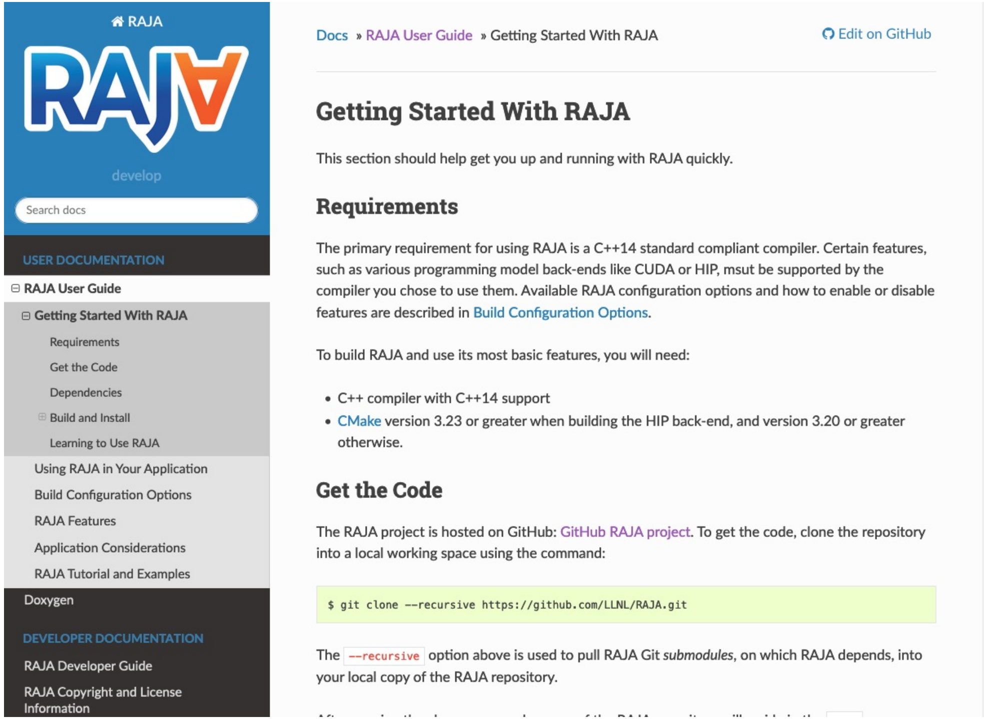


Contributions to develop, excluding merge commits and bot accounts



A variety of useful RAJA information is available

- **RAJA User Guide:** getting started info, details about features and usage, etc. (readthedocs.org/projects/raja)
- **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 RAJA team, vendors, DOE/ASC platform procurements, etc. (https://github.com/LLNL/RAJAPerf)



RAJA

develop

Search docs

USER DOCUMENTATION

⊟ RAJA User Guide

⊟ Getting Started With RAJA

Requirements

Get the Code

Dependencies

⊞ Build and Install

Learning to Use RAJA

Using RAJA in Your Application

Build Configuration Options

RAJA Features

Application Considerations

RAJA Tutorial and Examples

Doxygen

DEVELOPER DOCUMENTATION

RAJA Developer Guide

RAJA Copyright and License Information

Docs » RAJA User Guide » Getting Started With RAJA

Edit on GitHub

Getting Started With RAJA

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

Requirements

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

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

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

Get the Code

The RAJA project is hosted on GitHub: GitHub RAJA project. To get the code, clone the repository into a local working space using the command:

```
$ git clone --recursive https://github.com/LLNL/RAJA.git
```

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

These are linked on the RAJA GitHub project page.

RAJA and performance portability

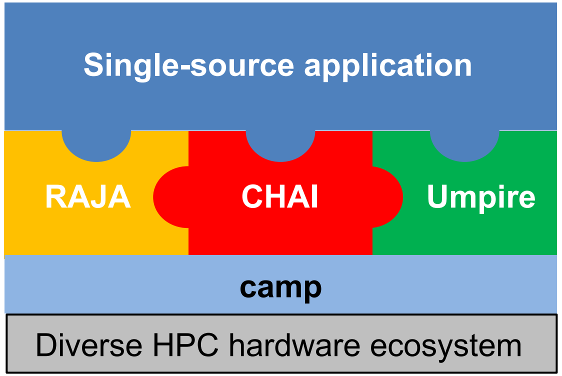
- RAJA is a **library of C++ abstractions** that enable users to write **portable, single-source** kernels – re-compile to run on different hardware architectures
  - Common HPC architectures: multicore CPUs, GPUs (NVIDIA, AMD, Intel), …
- RAJA **insulates application source code** from hardware and programming model-specific implementation details
  - OpenMP, CUDA, HIP, SYCL, 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'l data views for various access patterns, …
  - Loop tiling, thread-local data, GPU shared memory, …

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

The RAJA Portability Suite contains four complementary library projects

RAJA: C++ kernel execution abstractions

Enable <u>single-source</u> 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

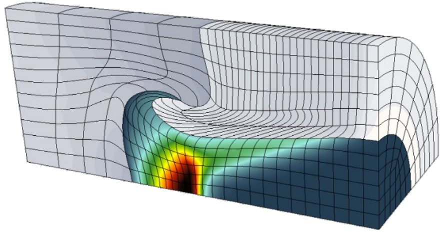
Focuses on HPC compiler compatibility and portability

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

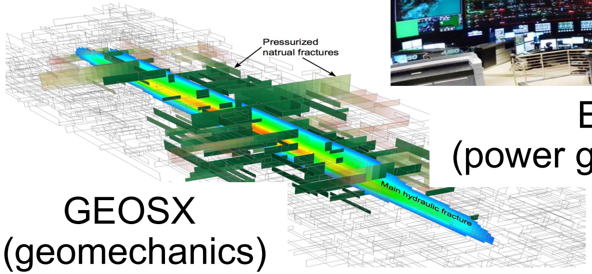
CHAI: C++ array abstractions

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

RAJA is used by ECP app and library projects and is a mission-critical dependency for most LLNL ASC codes

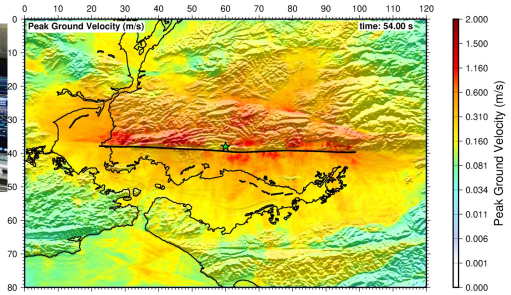


LLNL ATDM
(high-order FE ALE hydro)



GEOSX
(geomechanics)

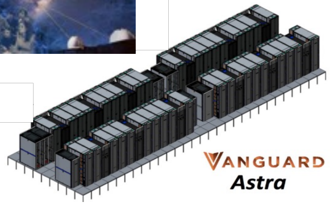
ExaSGD
(power grid optimization)



SW4
(earthquake modeling)

RAJA Portability Suite

Perlmutter (LBL)
AMD Milan CPU +
NVIDIA Ampere GPU



Astra (SNL)
ARM architecture

Sierra (LLNL)
IBM P9 CPU + NVIDIA Volta GPU

Aurora (ANL)
Intel Xeon CPU + Xe GPU

Frontier (ORNL) &
El Capitan (LLNL)
AMD CPU + GPU

RAJA is designed to promote usability and application developer productivity

- We want applications to maintain **single-source kernels** (as much as possible)
- RAJA provides benefits to application developers…
  - **Easy to understand and use** (especially those who are not CS or PM experts)
  - Allows **incremental and selective adoption** enabling experimentation
  - **Does not force major disruption** to app source code allowing custom wrapper layers
  - Promotes flexible algorithm implementations via **clean encapsulation**
  - Makes it **easy to parameterize kernel execution** via policy type aliases
  - Enables **systematic performance tuning** (tune classes of kernels, not individual kernels)

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

We will cover a variety topics today

- RAJA features:
  - **Simple loops**
  - **Reductions**
  - **Iteration spaces**
  - **Data layouts and views**
  - **Complex loops**
  - *Atomic, scan and sort operations (briefly)*
  - *More advanced features of the RAJA Portability Suite (briefly)*

- RAJA usage considerations (C++ templates and lambda expressions, memory management, etc.)

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 types defined in header files enable parameterization of loop execution

Simple Loops

C-style for-loop

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

Definitions placed in header files can be reused throughout application 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];
} );
```

Change the “execution policy” and/or “iteration space” to change the way a loop runs.

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

Simple Loops

C-style for-loop

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

Same loop body.

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];
} );
```

RAJA loop execution core concepts

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

- **Kernel execution template** specialized on **execution policy type** (sequential, OpenMP, CUDA, etc.)

RAJA loop execution core concepts

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

- Kernel execution template specialized on execution policy type (sequential, OpenMP, CUDA, etc.)
- **Iteration space object** (strided range, list of indices, etc.)

RAJA loop execution core concepts

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

- Kernel execution template specialized on execution policy type (sequential, OpenMP, CUDA, etc.)
- Iteration space object (strided range, list of indices, etc.)
- **Loop body** expressed 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::seq_exec

RAJA::omp_parallel_for_exec

RAJA::cuda_exec<BLOCK_SIZE>

RAJA::hip_exec<BLOCK_SIZE>

RAJA::sycl_exec<BLOCK_SIZE>
```

A sampling of RAJA loop execution policy types.

RAJA supports a variety of programming model back-ends to execute kernels

- Sequential (CPU)
- OpenMP multithreading (CPU)
- CUDA (NVIDIA GPUs)
- HIP (AMD GPUs)
- SYCL (Intel GPUs) – work-in-progress, not feature complete
- ”Vectorization” – SIMD (CPU), warp level vectorization (GPU)
- OpenMP target (target device such as a GPU) – available, but not considered production quality

Simple forall quiz!

- See **RAJA/exercises/vector-addition_solution.cpp** for an implementation of these code examples

C-style

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

Which RAJA execution policy emulates the C-style code?

`seq_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!

- See **RAJA/exercises/vector-addition_solution.cpp** for an implementation of these code examples

C-style

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

Which RAJA execution policy emulates the C-style code?
seq_exec
~~omp_parallel_for_exec~~
~~cuda_exec~~

RAJA-version

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

seq_exec executes sequentially as a basic C-style loop would

Simple forall quiz!

- See **RAJA/exercises/vector-addition_solution.cpp** for an implementation of these code examples

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-style code?

`seq_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!

- See **RAJA/exercises/vector-addition_solution.cpp** for an implementation of these code examples

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-style code?

~~seq_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!

- See **RAJA/exercises/vector-addition_solution.cpp** for an implementation of these code examples

C-style using CUDA directly

```
__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: Must 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),
  [=] __device__ (int i) {
    c[i] = a[i] + b[i];
  }
);
```

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

Before we continue, let's discuss a few C++ features essential to RAJA

Currently, RAJA requires C++14 or higher. In the next release, it will require C++17.

RAJA kernel execution methods are C++ templates

C++ features

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

- Templates allow you to write *generic* code and have the *compiler* generate an implementation for the set of template parameter types you specify
- Here, **`ExecPol`**, **`IdxType`**, and **`LoopBody`** are C++ types you provide in your code. They must be known at <u>compilation time.</u>

RAJA kernel execution methods are C++ templates

C++ features

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

- You specify **`ExecPol`**, **`IdxType`**, and **`LoopBody`** types ...

Like this...

```
forall< seq_exec >( TypedRangeSegment<int>(0, N),
  [=] (...) {
    // loop body
  }
);
```

- Note: **`IdxType`** and **`LoopBody`** types are deduced by the compiler based on arguments you give to the **`forall`** method

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

C++ features

This thing…

```
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 <u>more concise and easier to use</u>

C++ 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 (outer scope) variables are pulled into the lambda data environment
  - We recommend using **capture by-value** for portable code… **`[=]`**
- The parameter list is a set of arguments passed to the lambda – **`(int i)`** is a loop index
- A lambda passed to a GPU kernel requires a device annotation, such as [=] **__device__** (…) { … }

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

"Bring your own" memory management

- RAJA does not provide a memory model. This is by design.
  - Our users prefer to handle memory space allocations and transfers

```
forall< cuda_exec<256> >(range, [=] __device__ (int i) {
  a[i] = b[i];
} );
```

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

"Bring your own" memory management

```
forall< cuda_exec<256> >(range, [=] __device__ (int i) {
  a[i] = b[i];
} );
```

'a' and 'b' must be accessible on GPU!!

- Some possibilities for moving data between CPU and GPU memory:
  - **Manual** – e.g., use cudaMalloc( ), cudaMemcpy( ) to allocate, copy to/from device
  - **Unified Memory (UM)** – e.g., use cudaMallocManaged( ) for paging on demand
  - **Umpire** – a uniform interface that is the same regardless of system and memory type
  - **CHAI** – C++ array abstraction that automates data copies as needed (see github.com/LLNL/CHAI)

CHAI and Umpire are part of the RAJA Portability Suite.

Umpire provides portable memory management tools for HPC applications

Umpire Concepts

- A **Memory Resource** is a kind of memory, with specific performance and accessibility characteristics
- A **ResourceManager** is the top-level interface for building resource allocators, moving data, etc.
- An **Allocation Strategy** decouples how and where allocations are made
- An **Allocator** is a lightweight interface for making an allocation and querying it (same for all resources)
- An **Operation** manipulates data in memory (copy, move, reallocate, memset, etc.)

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

auto d_pool =
  rm.makeAllocator<QuickPool>("MY_POOL", d_alloc);


void* h_data = h_alloc.allocate(1024);
void* d_data = d_pool.allocate(1024);

rm.memset(h_data, 0);
rm.copy(d_data, h_data);

h_alloc.deallocate(h_data);
```

With Umpire, memory operations look the same regardless of the underlying memory system & programming model

```
auto& rm = umpire::ResourceManager::getInstance();

auto h_alloc = rm.getAllocator("HOST");
auto d_alloc = rm.getAllocator("DEVICE");
auto d_pool = rm.makeAllocator<umpire::strategy::QuickPool>("POOL", d_alloc);

double* a_h = static_cast<double *>(h_alloc.allocate( N*sizeof(double) ));
double* b_h = static_cast<double *>(h_alloc.allocate( N*sizeof(double) ));
double* a_d = static_cast<double *>(d_pool.allocate( N*sizeof(double) ));
double* b_d = static_cast<double *>(d_pool.allocate( N*sizeof(double) ));

// init a_h, b_h on CPU (host)

rm.copy(a_d, a_h);
rm.copy(b_d, b_h);

RAJA::forall< RAJA::cuda_exec<256> >(..., [=] __device__ (int i) { a_d[i] += b_d[i]; } );

rm.copy(a_h, a_d);
// do something with a_h on CPU...
```

A device memory pool is more efficient than doing standard device allocations & deallocations

Copy operations expressed with pointers only

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 encapsulate 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 reduction 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 because different threads would see different partial reduction values while the kernel executes.

- 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::hip_reduce

RAJA::sycl_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 reductions can be performed 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) {
    sum += a[i];

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

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

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

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

	0	1	2	. . .					N/2	. . .					N-1
a :	1	-1	1	-1	1	. . .	1	-10	10	-10	1	. . .	-1	1	-1

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

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

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

	0	1	2	. . .					N/2	. . .				N-1
a :	1	-1	1	-1	1	. . .	1	-10	10	-10	1	. . .	-1	1	-1

- *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 may vary from run to run.

We are developing a new reduction interface (more flexible, better PM integrability)

Current reduction API

```
RAJA::ReduceSum<REDUCE_POL, int> sum(0);
RAJA::ReduceMin<REDUCE_POL, int> min(MAX);

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

    sum += a[i];
    min.min(a[i]);
  }
);

int my_sum = sum.get();
int my_min = min.get();
```

New reduction API

```
int sum(0);
int min(MAX);

RAJA::forall<EXEC_POL>( range,
  RAJA::expt::Reduce<RAJA::operators::plus>(&sum),
  RAJA::expt::Reduce<RAJA::operators::min>(&min),
  [=](int i, int& _sum, int& _min) {

    _sum += a[i];
    _min = RAJA_MIN(a[i], _min);
  }
);

int my_sum = sum;
int my_min = min;
```

1. No special reduction objects.
2. Reduction ops passed to kernel exec method & reduction vars passed to lambda.

- max, minloc, and maxloc also available (not shown here). See RAJA User Guide.
- Available with RAJA::forall. Support for other RAJA kernel execution methods on the way.

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)

CPU and GPU RAJA variants look the same, except for the policies; e.g.
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

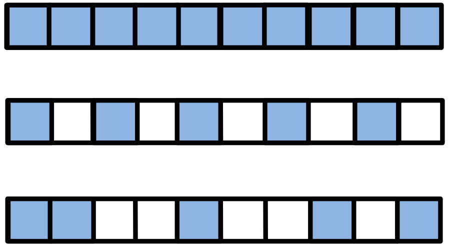
Iteration spaces

- A **Segment** encapsulates a set of loop indices to run in a kernel, such as

Contiguous range [beg, end)

Strided range [beg, end, stride)

List of indices (indirection)



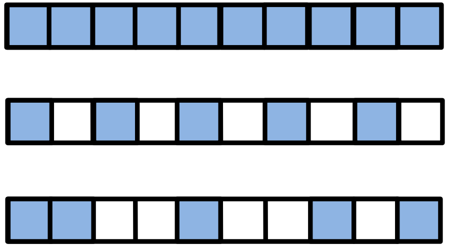
Loop iteration spaces are defined by Segments

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

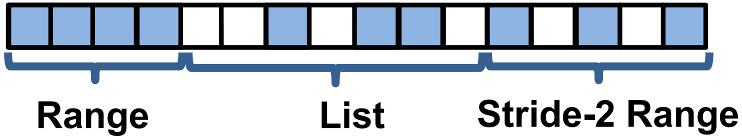
Contiguous range [beg, end)

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)

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

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

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

A RangeStrideSegment defines a strided sequence of indices

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

```
RAJA::forall< RAJA::seq_exec >( srange1 , [=] (int i)
{
    y[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> srange2( ?, N, ? );

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

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

A RangeStrideSegment defines a strided sequence of indices

Iteration spaces

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

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

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

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) {
    y[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

```
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

```
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 directly in loop body code.

IndexSets enable iteration space partitioning

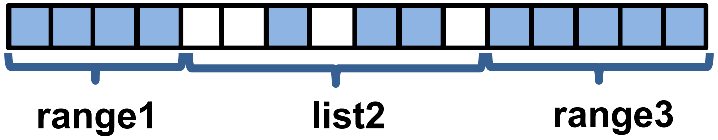
```
RAJA::TypedIndexSet< RangeSegType, ListSegType > iset;
```

A
```
RangeSegType range1(0, 8);
```

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

C
```
RangeSegType range3(24, 28);
```

```
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
A		B		C

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 and accessed 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

- 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

- 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 code 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 basic Layout<DIM> type.

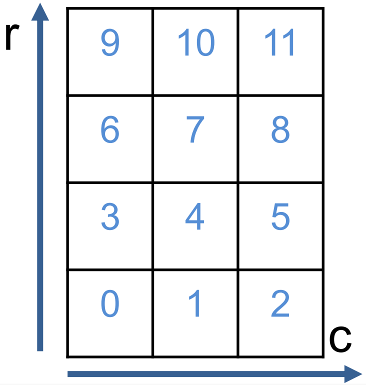
Every Layout has a permutation

```
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).

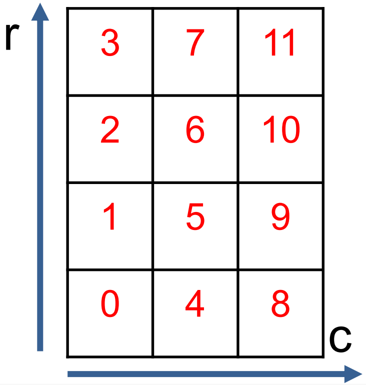
Every Layout has a permutation

```
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).

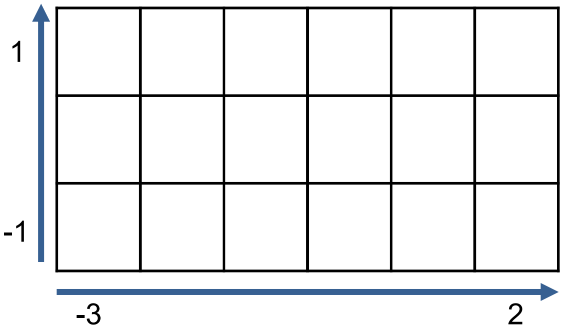
Offset layout quiz…

```
RAJA::OffsetLayout<2> offset_layout =
  RAJA::make_offset_layout<2>( {{-1, -3}}, {{2, 3}}
);
```

- *What index space does this layout represent?*

Offset layout quiz…

```
RAJA::OffsetLayout<2> offset_layout =
  RAJA::make_offset_layout<2>( {{-1, -3}}, {{2, 3}}
);
```

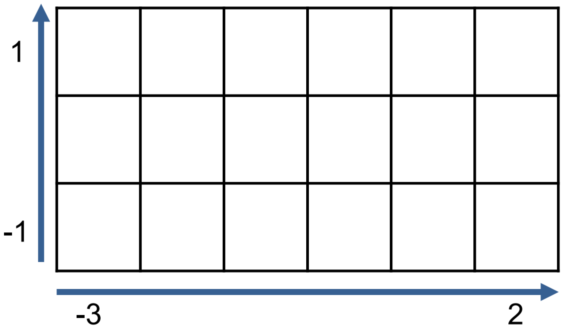


- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

Offset layout quiz…

```
RAJA::OffsetLayout<2> offset_layout =
  RAJA::make_offset_layout<2>( {{-1, -3}}, {{2, 3}}
);
```



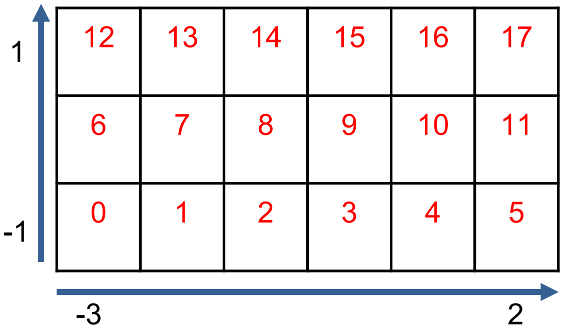
- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

- *Which index is stride-1?*

Offset layout quiz…

```
RAJA::OffsetLayout<2> offset_layout =
  RAJA::make_offset_layout<2>( {{-1, -3}}, {{2, 3}}
);
```



- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

- *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, -3}}, {{2, 3}}
);
```

1

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

-1

-3 2

- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

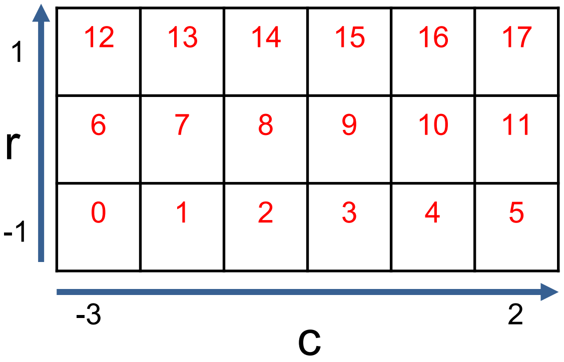
- *Which index is stride-1?*

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, -3}}, {{2, 3}}
);
```



- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

- *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 6, since the right-most index has extent 6, [-3, 3).

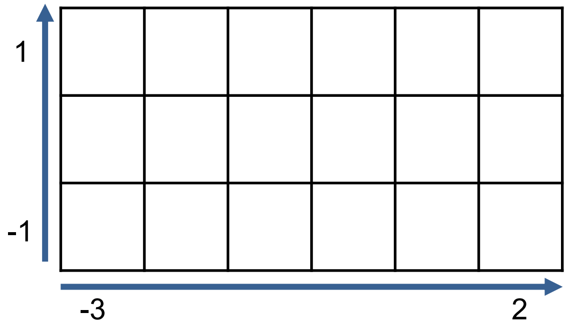
Let's try a permuted offset layout…

```
std::array<RAJA::idx_t, 2> perm {{1, 0}};
RAJA::OffsetLayout<2> permoffset_layout =
  RAJA::make_permuted_offset_layout<2>( {{-1, -3}}, {{2, 3}}, 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, -3}}, {{2, 3}}, perm );
```

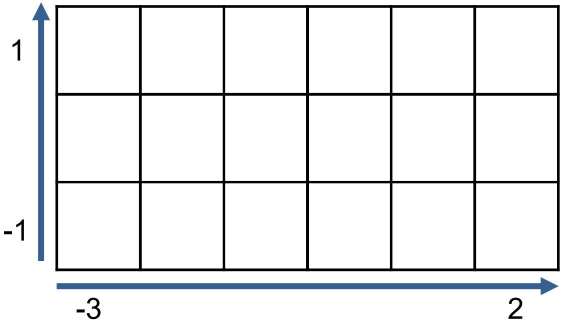


- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

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, -3}}, {{2, 3}}, perm );
```



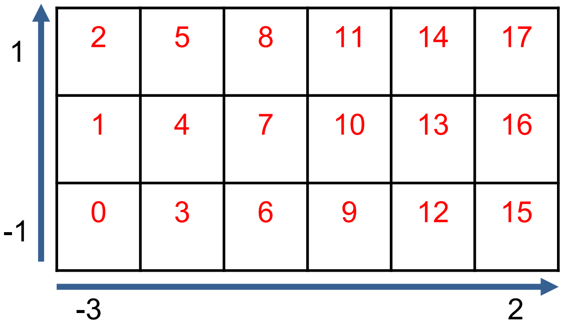
- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

- *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, -3}}, {{2, 3}}, perm );
```



- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

- *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, -3}}, {{2, 3}}, perm );
```

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

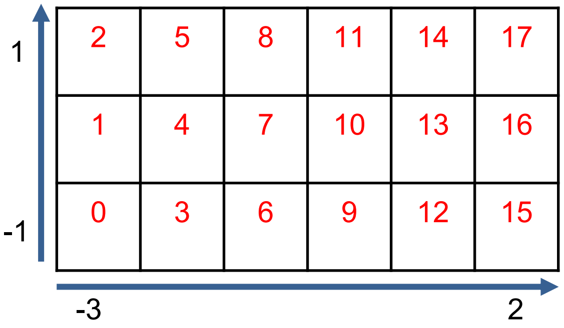
- *Which index is stride-1?*

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, -3}}, {{2, 3}}, perm );
```



- *What index space does this layout represent?*

The 2-d index space [-1, 2) x [-3, 3).

- *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 the RAJA ‘Launch’ API

We illustrate the RAJA Launch API using a nested loop kernel for matrix multiplication

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

   C(row, col) = dot;

  }
}
```

The inner loop body is expressed as the function body of a lambda expression

Nested loops

```
// 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(row, k)* B(k, col);
   }

   C(row, col) = dot;

  }
}
```

```
launch<launch_policy>(
  LaunchParams(Teams(NTeams), Threads(NThreads)),
  [=] (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;

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

‘Loop’ methods replace for-loops

Nested loops

```
// 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(row, k)* B(k, col);
   }

   C(row, col) = dot;

  }
}
```

```
launch<launch_policy>(
  LaunchParams(Teams(NTeams), Threads(NThreads)),
  [=] (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;

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

The entire loop structure is passed as a lambda expression to a ‘launch’ execution space

Nested loops

```
// 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(row, k)* B(k, col);
   }

   C(row, col) = dot;

  }
}
```

```
launch<launch_policy>(
  LaunchParams(Teams(NTeams), Threads(NThreads)),
  [=] (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;

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

The ‘launch’ method creates an execution space with a given teams & threads configuration.

Launch quiz!

RAJA

What are the values of foo and bar after completion of the loops?

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

  int foo = 0, bar = 1;

  loop<seq_policy>(ctx, RangeSegment(0,4), [&](int ii){

    foo = ii * bar;

    loop<seq_policy>(ctx, RangeSegment(ii,3), [&](int jj){

      bar += jj;

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

Launch quiz!

Nested loops

RAJA

What are the values of foo and bar after completion of the loops?

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

  int foo = 0, bar = 1;

  loop<seq_policy>(ctx, RangeSegment(0,4), [&](int ii){

    foo = ii * bar;

    loop<seq_policy>(ctx, RangeSegment(ii,3), [&](int jj){

      bar += jj;

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

C-style

```
int foo = 0, bar = 1;
for (int ii = 0; ii < 4; ++ii ){

  foo = ii * bar;

  for (int jj = ii; jj < 3; ++jj ){

    bar += jj;

  }
}
```

Launch quiz!

Nested loops

RAJA

What are the values of `foo` and `bar` after completion of the loops?

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

  int foo = 0, bar = 1;

  loop<seq_policy>(ctx, RangeSegment(0,4), [&](int ii){

    foo = ii * bar;

    loop<seq_policy>(ctx, RangeSegment(ii,3), [&](int jj){

      bar += jj;

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

C-style

```
int foo = 0, bar = 1;
for (int ii = 0; ii < 4; ++ii ){

  foo = ii * bar;

  for (int jj = ii; jj < 3; ++jj ){

    bar += jj;

  }
}
```

`bar` = 1 + 0 + 1 + 2 + 1 + 2 + 2 = 9

`foo` = 3 * 9 = 27

Launch quiz!

RAJA

What are the values of foo and bar after completion of the loops?

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

  int foo = 0, bar = 1;

  loop<seq_policy>(ctx, RangeSegment(0,4), [&](int ii){

    foo = ii * bar;

    loop<seq_policy>(ctx, RangeSegment(ii,3), [&](int jj){

      bar += jj;

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

C-style

```
int foo = 0, bar = 1;
for (int ii = 0; ii < 4; ++ii ){

  foo = ii * bar;

  for (int jj = ii; jj < 3; ++jj ){

    bar += jj;

  }
}
```

bar = 1 + 0 + 1 + 2 + 1 + 2 + 2 = 9
foo = 3 * 9 = 27

RAJA::launch can handle imperfectly nested loops and loop carried dependencies.

RAJA launch uses a thread/team model similar to the CUDA/HIP thread/block model


```
launch<launch_policy>(
  LaunchParams(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;
    });
  });
});
```

Matrix-Matrix multiplication kernel

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

Loops can be mapped to CUDA/HIP threads or blocks

Selecting host or device execution at run time is enabled by providing both host and device policies

Nested loops

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

```
launch<launch_policy>( cpu_or_gpu,
  LaunchParams(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;
    });
  });
});
```

Matrix-Matrix multiplication kernel

```
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>;
```

Methods and types in RAJA namespace

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

Nested loops

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

using col_policy =
LoopPolicy<host_policy, cuda_thread_x_loop>;

launch<launch_policy>(
  LaunchParams(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;
    });
  });
});
```

Matrix-Matrix multiplication kernel with block/thread policy

```
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;

}
```

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

Nested loops

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

using col_policy =
LoopPolicy<host_policy, cuda_thread_x_loop>;

launch<launch_policy>(
  LaunchParams(Teams(???), 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;
    });
  });
});
```

Matrix-Matrix multiplication kernel with block/thread policy

```
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;

}
```

How many `Teams()` do we need for this problem?

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

Nested loops

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

using col_policy =
LoopPolicy<host_policy, cuda_thread_x_loop>;

launch<launch_policy>(
  LaunchParams(Teams(N), 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;
    });
  });
});
```

Matrix-Matrix multiplication kernel with block/thread policy

```
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;

}
```

How many `Teams()` do we need for this problem?

`Teams(N)` - one block for every row.

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

Nested loops

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

using col_policy =
LoopPolicy<host_policy, cuda_thread_x_loop>;

launch<launch_policy>(
  LaunchParams(Teams(N), Threads(32)),
  [=] 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;
    });
  });
});
```

Matrix-Matrix multiplication kernel with block/thread policy

```
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;

}
```

What happens if `Threads(32)` is set in the launch parameters?

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

Nested loops

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

using col_policy =
LoopPolicy<host_policy, cuda_thread_x_loop>;

launch<launch_policy>(
  LaunchParams(Teams(N), Threads(32)),
  [=] 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;
    });
  });
});
```

Matrix-Matrix multiplication kernel with block/thread policy

```
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;

}
```

What happens if `Threads(32)` is set in the launch parameters?

`Threads(32)` sets the number of threads per block (`blockDim.x`) to 32 so that `dot` accumulates products of each row in 32 column chunks.

Global thread ID calculations are simplified with RAJA policies

Nested loops

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

using col_policy =
LoopPolicy<host_policy, cuda_global_thread_x>;

launch<launch_policy>(
  LaunchParams(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;
    });
  });
});
```

Matrix-Matrix multiplication kernel with global threads

```
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;
}
```

Brief note on nested loops with the RAJA ‘Kernel’ API

RAJA 'kernel' is an alternative API that encodes all loop structure & execution in an execution policy

Nested loops

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

   C(row, col) = dot;

 }
}
```

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

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;
  }
);
```

Methods and types are in RAJA namespace

Source files for these examples and others that compare ‘kernel’ and ‘launch’ are in the RAJA repo

4 exercises are implemented with both Kernel and Launch to compare APIs. Please look in the RAJA/exercises/ directory.	
kernelintro-execpols.cpp kernelintro-execpols_solution.cpp	launchintro-execpols.cpp launchintro-execpols_solution.cpp
kernel-matrix-transpose.cpp kernel-matrix-transpose_solution.cpp	launch-matrix-transpose.cpp launch-matrix-transpose_solution.cpp
kernel-matrix-transpose-tiled.cpp kernel-matrix-transpose-tiled_solution.cpp	launch-matrix-transpose-tiled.cpp launch-matrix-transpose-tiled_solution.cpp
kernel-matrix-transpose-local-array.cpp kernel-matrix-transpose-local-array_solution.cpp	launch-matrix-transpose-local-array.cpp launch-matrix-transpose-local-array_solution.cpp

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 <u>must be compatible</u> with the loop execution policy (similar to reductions).

Scan: RAJA provides a prefix-sum scan operation by default

```
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' arrays have length N. '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 "+".

An optional 3rd argument is an operator to support other scan operations, such as '<', '>', etc.

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).

Other capabilities provided by the RAJA Portability Suite which may be of interest

Shared or stack local memory can be accessed by all threads in RAJA kernels

Local Data

```
launch<launch_policy>(
  LaunchParams(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.

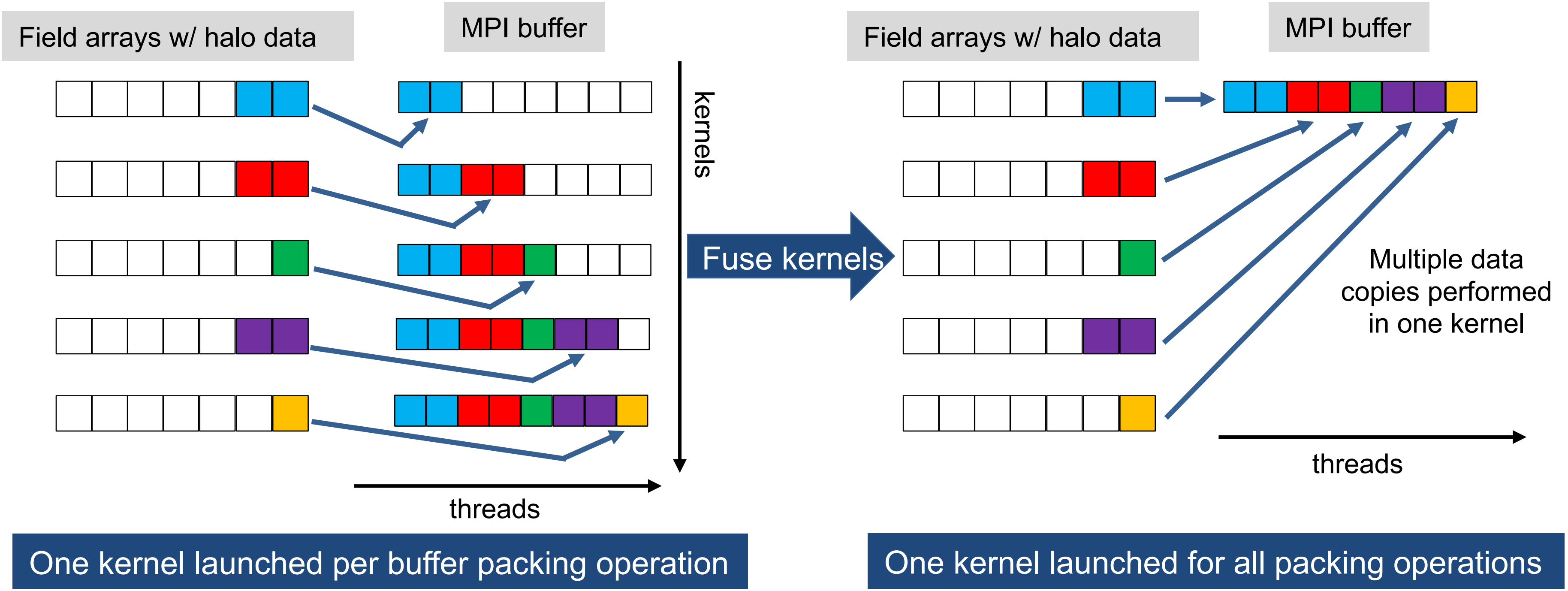
On the GPU, this is a shared memory array which can be accessed by all threads within a Team (block).

A dynamic version of shared memory is also available.

An analogous capability 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



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, buffers[neighbor]);
}
```

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

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, buffers[neighbor]);
}
```

Application considerations

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

- Profile your code to see where performance is most important
  - Do a few/no kernels dominate runtime?
  - Can you afford to maintain multiple, (highly-optimized) architecture-specific versions of important kernels?

- 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?

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
- Assign execution policies to **“classes of loops/kernels”**
  - 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

- Application coding styles may need to change regardless of programming model; e.g., to get good GPU performance
  - Change algorithms as needed to ensure correct parallel execution
  - Move variable declarations to innermost scope to avoid thread correctness 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?

Lawrence Livermore National Laboratory