EXASCALE COMPUTING PROJECT

RAJA: A Technical Perspective

NERSC GPUs for Science Day

July 2-3, 2019

David Beckingsale

LLNL-PRES-777737

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

RAJA and performance portability

- RAJA is a **library of C++ abstractions** that allow you to write single-source loop kernels that can be run on different platforms by re-compiling your code
  - Multicore CPUs, Xeon Phi, NVIDIA GPUs, …
- RAJA helps you **insulate your code** from hardware and programming model-specific implementation details
  - SIMD vectorization, OpenMP, CUDA, …
- RAJA supports many **parallel patterns** and **performance tuning** options
  - Simple and complex loop kernels
  - Reductions, scans, atomic operations
  - Loop tiling, thread-local data, GPU shared memory, …

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

RAJA design goals focus on usability and developer productivity

- Applications should maintain **single-source kernels** (if possible)
- **Easy to understand** 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**

RAJA is developed collaboratively with production application teams.

A loop written with a standard programming language exposes all aspects of execution explicitly

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

C-style for-loop

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

In the implementation, loop iteration order, data access, etc. are explicit in source code.

RAJA encapsulates execution details so a loop can run differently without changing source code

C-style for-loop

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

RAJA-syle loop

```
using EXEC_POL = ...;

RAJA::RangeSegment range(0, N);

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

RAJA encapsulates execution details so a loop can run differently without changing source code

C-style for-loop

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

RAJA-syle loop

```
using EXEC_POL = ...;

RAJA::RangeSegment range(0, N);

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

Typically, these are defined in a header file.

Writing a loop with RAJA requires a change to the loop header, but body typically is unchanged.

RAJA loop execution consists of four core concepts

```
using EXEC_POLICY = ...;
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)

RAJA loop execution core concepts

```
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.)

RAJA loop execution core concepts

```
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.)

RAJA loop execution core concepts

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

The programmer must make sure the loop body works with the chosen execution policy; e.g., thread safety.

- 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

By changing the execution policy, you change the way the loop will run

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

```
RAJA::simd_exec

RAJA::omp_parallel_for_exec

RAJA::cuda_exec<BLOCK_SIZE>

RAJA::omp_target_parallel_for_exec<MAX_THREADS_PER_TEAM>

RAJA::tbb_for_exec
```

Examples of RAJA loop execution policy types.

“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?

“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?

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

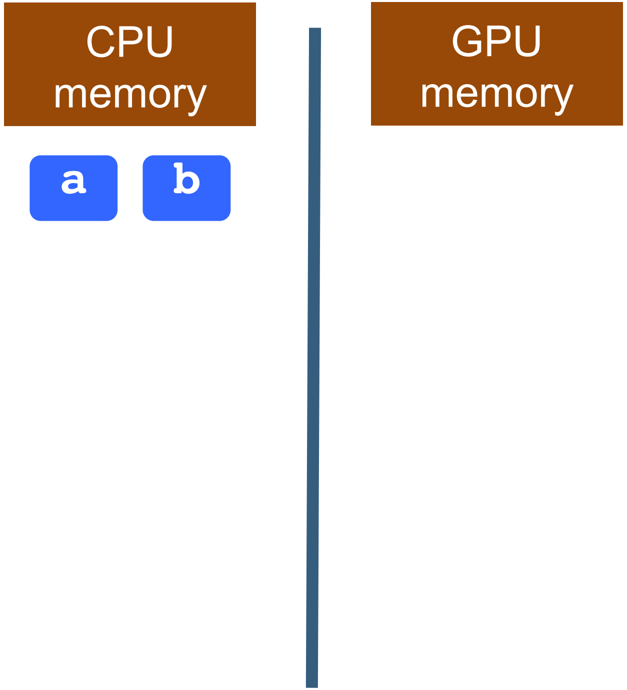
CHAI was developed to complement to RAJA.

CHAI provides array abstractions for transparent, automatic data copies

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

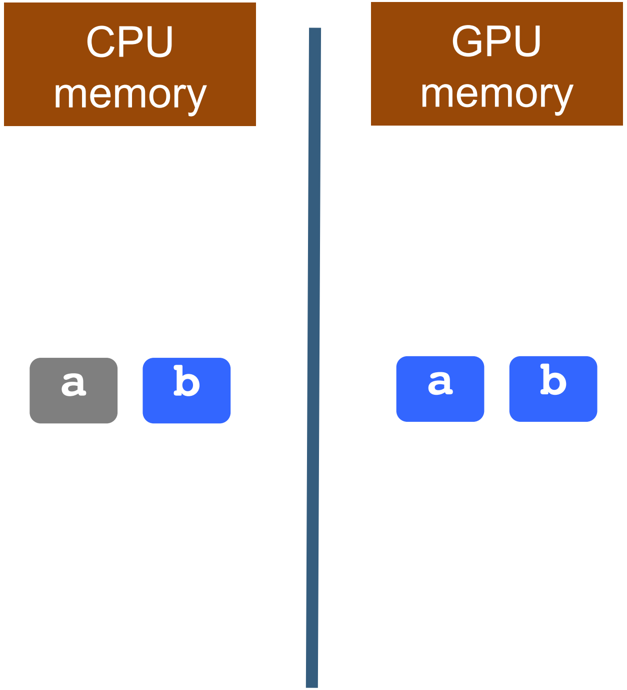


CHAI provides array abstractions for transparent, automatic data copies

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

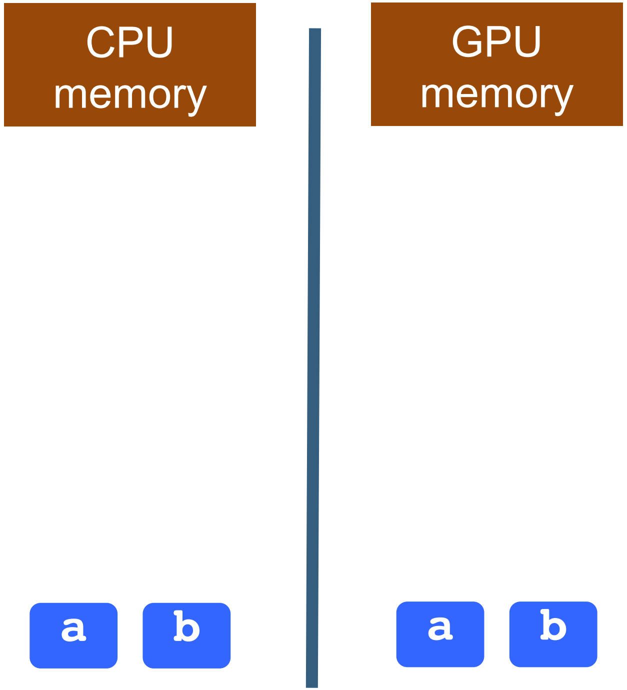


CHAI provides array abstractions for transparent, automatic data copies

```
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.

Recent RAJA development has focused on complex kernels, multi-dimensional data, and advanced execution features

Matrix transpose kernel (C-style)

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

    At[row + N*col] = A[col + N*row];

  }
}
```

```
using KERNEL_POL = ... ;

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

      Atview(col, row) = Aview(row, col)

  }
);
```

Multiple iteration spaces & lambda arguments

RAJA Views enable flexible indexing (see RAJA user guide)

Change execution policy, not kernel code, to change how loop runs; e.g.,

- Permute loop levels
- OpenMP variations, including collapse
- CUDA kernel block-thread mapping variations
- Tiled loops (cache-blocking, GPU shared memory)

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

The RAJA::kernel interface uses four basic concepts that are analogous to those with RAJA::forall

```
using KERNEL_POL = ... ;

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

1. Kernel **execution template** (‘RAJA::kernel’)
2. Kernel **execution polic<u>ies</u>** (in ‘KERNEL_POL’)
3. Kernel **iteration spac<u>es</u>** (e.g., ‘RangeSegments’)
4. Kernel **body** (lambda expression<u>s</u>)

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

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

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

```
Lambda<0>
```

```
CudaSyncThreads
```

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

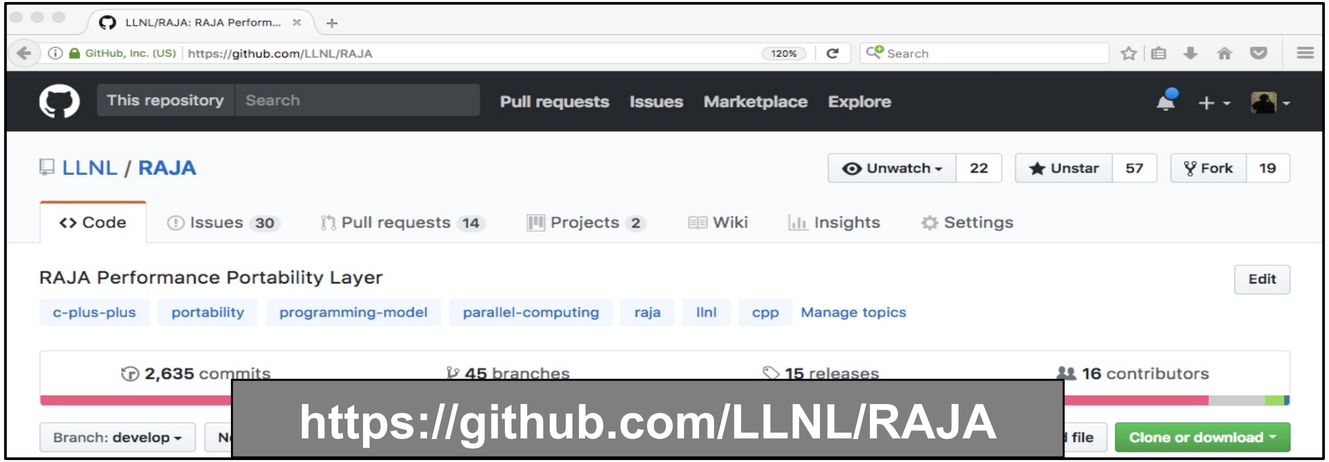
```
For<0, exec_policy0,
   Lambda<0>,
   For<2, exec_policy2,
      Lambda<1>
   >
>
```

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

RAJA supports a variety of parallel constructs and loop patterns

- Simple and complex loop patterns
  - Non-perfectly nested loops
  - Loop tiling
- Kernel transformations (via **execution policy changes**)
  - Change order of loop iterations
  - Permute loop nest ordering
  - Multi-dimensional data views with offsets and index permutations
  - Direct CUDA thread-block mapping control
  - CPU/GPU shared and thread local memory
- Portable reductions, scans, and atomic operations
- Multiple execution back-ends: sequential, SIMD, OpenMP (CPU, target offload), CUDA, AMD HIP (in progress), Intel Threading Building Blocks (experimental)

RAJA is an open source project developed by CS researchers, app developers, and vendors



- User Guide & Tutorial: https://readthedocs.org/projects/raja/
- RAJA Performance Suite: https://github.com/LLNL/RAJAPerf

RAJA is supported by LLNL programs (ASC and ATDM) and the ECP (ST).

Acknowledgements

RAJA Team and contributors

- Rich Hornung (PL)
- David Beckingsale
- Jason Burmark
- Noel Chalmers (AMD)
- Robert Chen
- Matt Cordery (IBM)
- Chip Freitag (AMD)
- Jeff Hammond (Intel)
- Holger Jones
- Jeff Keasler
- Will Killian (Millersville University)
- Adam Kunen
- Scott Moe (AMD)
- Olga Pearce
- Tom Scogland
- Arturo Vargas

Plus, all the application developers, Livermore Computing staff, vendor contributors and compiler teams, and others who have helped make these projects more robust and viable for production codes.

Lawrence Livermore National Laboratory