An Overview of RAJA

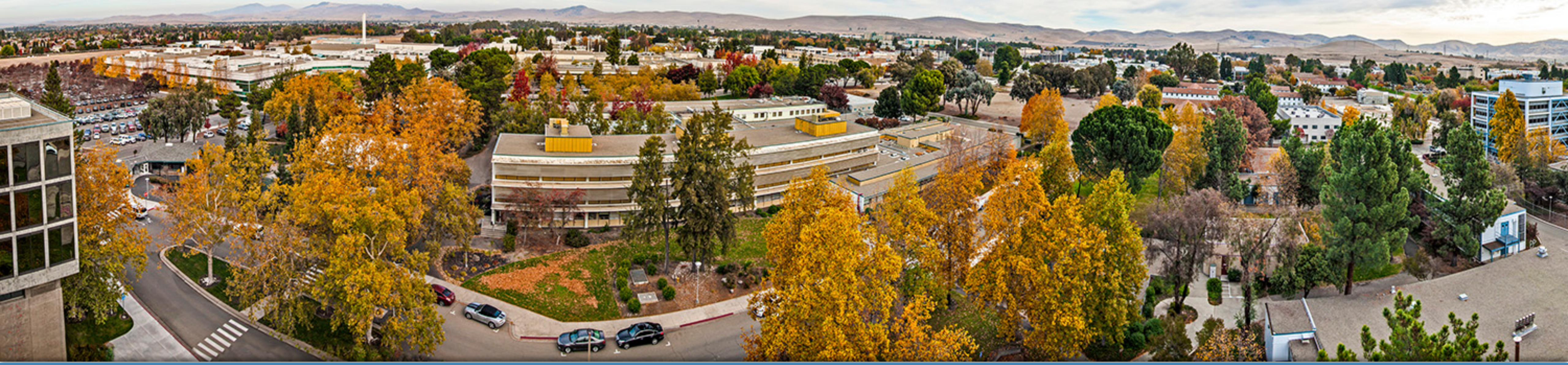
ALCF Computational Performance Workshop

May 5, 2021

EXASCALE COMPUTING PROJECT

Presentation by: Brian Homerding

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



LLNL-PRES-781841
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

RAJA and performance portability

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

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

RAJA loop execution has four core concepts

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

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

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

Simple loops

RAJA loop execution core concepts

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

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

RAJA loop execution core concepts

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

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

These core concepts are common threads throughout our discussion

Simple loops

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

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

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

The execution policy determines the programming model back-end

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

```
RAJA::simd_exec

RAJA::omp_parallel_for_exec

RAJA::cuda_exec<BLOCK_SIZE, Async>

RAJA::omp_target_parallel_for_exec<MAX_THREADS_PER_TEAM>

RAJA::tbb_for_exec
```

A sample of RAJA loop execution policy types.

Nested loops

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

Nested loops

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

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

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

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

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

Each loop level has an iteration space and loop variable

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

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

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

Each loop level has an execution policy

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

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

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

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

Nested loops

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

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

'For' statements are swapped.

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

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

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

Loop tiling

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

Loop tiling

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

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

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

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

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

      }
    }
  }
}
```

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

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

Loop tiling

```
using namespace RAJA;

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

          ...
        >
      >
  >;
```

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

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

Loop tiling

```
using namespace RAJA;

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

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

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

          >
        >
      >
    >
  >;
```

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

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

(Thread) local data

Sometimes kernels require multiple lambdas to fully describe implementation

Local data

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

- Again, recall the matrix multiplication example:

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

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

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

Use lambda statements to define intervening code between loops

Local data

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

    double dot = 0.0;

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

    C(row, col) = dot;

  }
}
```

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

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

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

Local data

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

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

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

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

  ...

);
```

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

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

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

Local data

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

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

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

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

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

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

Policy example: collapse loops in an OpenMP parallel region

Local data

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

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

    >
  >;
```

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

Local data

Policy example: launch loops as a CUDA kernel

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

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

        >
      >
    >
  >;
```

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

Materials that supplement this presentation are available

Wrap up

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