RAJA Tutorial: Hands-on Session

LLNL internal RAJA tutorial

April 15, 2020

EXASCALE COMPUTING PROJECT

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

Hands-on session

- Earlier, we described RAJA and how it enables performance portability
- In this session, you have an opportunity to try out RAJA and work with it
  - You can work through exercises we provide
  - Or, you can try it out in your own code
- We have multiple RAJA developers in the room to assist you as needed

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

Please don’t hesitate to ask a question at any time

Setting up…

What you need to do to get started depends on what you will work on today

- If you will work through our exercises on an LC machine, you will need to get the RAJA code and build it in your workspace
- If you prefer to work on your laptop, we have a Docker container that you can use that has RAJA in it already built (Mac OSX only!)
- If you will work on your own code, you will need to get the RAJA code, build and install it in your workspace, and then compile and link against it with your build system

We will walk through these processes next.

Getting the code (not using Docker container)

- Clone the RAJA repository into your working space (LC or laptop) by typing:

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

Getting the code (not using Docker container)

- Clone the RAJA repository into your working space (LC or laptop) by typing:

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

This is needed to get all necessary git submodules

Getting the code (not using Docker container)

- Clone the RAJA repository into your working space (LC or laptop) by typing:

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

- By default, you will be on the 'develop' branch, which is fine for this tutorial or working on your own code. If you change to another branch at some point, you will need to run:

```
git submodule update --recursive
```

Getting the code (not using Docker container)

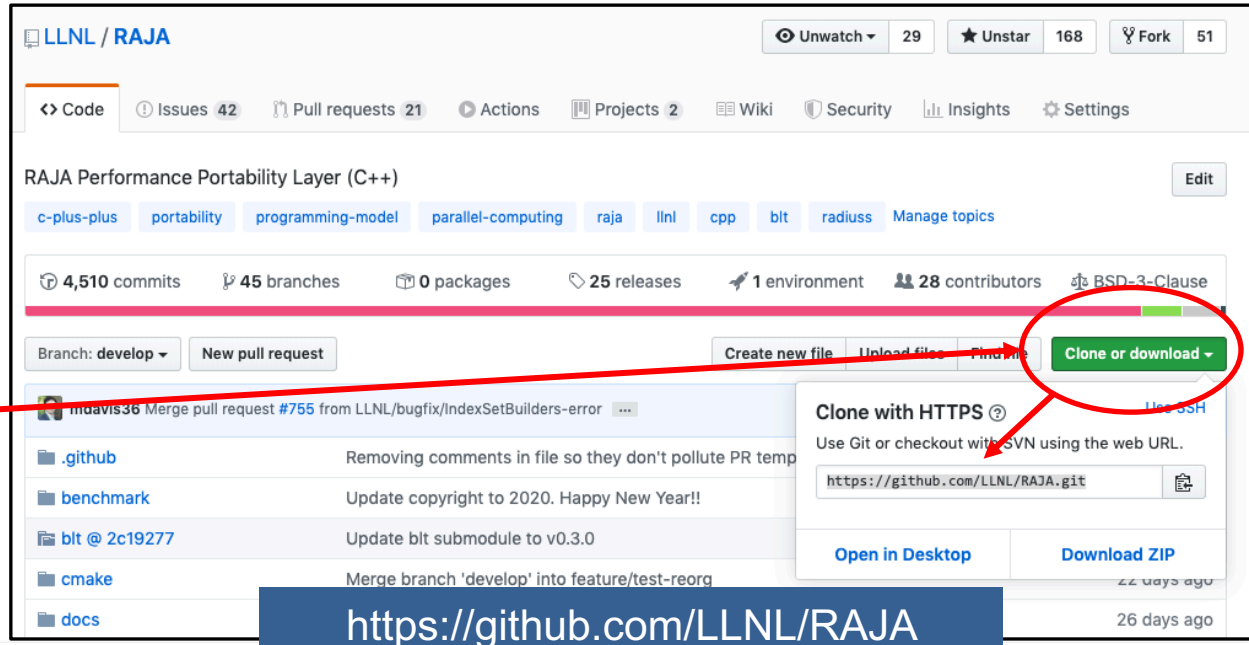
- Clone the RAJA repository into your working space (LC or laptop) by typing:

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

- If you change branches, you may need to run:

```
git submodule update --recursive
```

You can always get the URL on the RAJA GitHub project page by clicking here



https://github.com/LLNL/RAJA

Building RAJA on an LC machine

- The RAJA repo contains build scripts you can use to build the code on an LC machine. Look in:

  `RAJA/scripts/lc-builds`

- We recommend using those to get started. For example, to use clang9 on a TOSS3 machine, run the following in the top-level RAJA directory:

  `./scripts/lc-builds/toss3_clang9.0.0.sh`

  This will create a directory called:

  `build_lc_toss3-clang-9.0.0`

- Then, to compile RAJA, its tests and exercises, ”cd” into that directory and run:

  `make` or **`make –j <#>`** (for a parallel build)

Building RAJA on a non-LC machine

- You will have to invoke CMake appropriately before building with ‘make’
- Typically, it goes something like the following. Starting in the top-level RAJA directory:

```
> mkdir my-build && cd my-build

> cmake < options > ../

> make   (or make –j <#>)
```

- The RAJA build scripts mentioned on the previous slide may be helpful for you to figure out which options you should pass to CMake (e.g., compiler flags, install directory, etc.). Look in:

RAJA/scripts/lc-builds

If you prefer to work on your laptop (Mac only!)

- We have a Docker container you can use to work our exercises (supports CPU execution only).
- You need to have Docker Desktop (or similar) installed so you can execute Docker commands. For example, see https://docs.docker.com/install

  — Get the RAJA tutorial Docker container by running the following command:

  docker run -it rajaorg/raja-tutorial:radiuss2020

  — This puts you in a bash terminal inside the Docker container, which already has RAJA built

  — You can use vi or emacs to edit the exercise files

Working RAJA tutorial exercises is the same whether you are on the LC or your (Mac) laptop

- Files are located in the directory:

  `RAJA/exercises/tutorial_halfday`

- We will discuss each exercise one at a time, describing what you are to do
- Each exercise has two files; e.g., 'ex1_foo.cpp' and 'ex1_foo_solution.cpp'
- To work through an exercise….
  - The associated exercise file describes the exercise. Edit it and insert RAJA code as requested (locations are indicated by comments containing the text 'TO DO…' and 'EXERCISE')
  - Compile the code (i.e., run 'make' in your build directory)
  - Run the exercise executable file (i.e., located in 'build/bin' directory)
  - Screen output will indicate pass or fail. Adjust your attempt as needed. Compare to 'solution' file if needed.

If you have a question, please don't hesitate to ask.

If you will be working on your own code

- You need to install RAJA after building it. Run the following command in your build directory after compiling:

```
make install
```

- If you are working on an LC machine, and you used a RAJA build script to build to code, you will see that an install directory has been created; e.g.,

```
install_lc_toss3-clang-9.0.0
```

- This directory contains an ‘include’ and a ‘lib’ directory that you will use
- The RAJA “template project” contains information that can help you compile and link against an installed version of RAJA. Please see:

https://github.com/LLNL/RAJA-project-template

Exercise #1:
Simple loop execution

Recall the basic RAJA loop transformation

C-style for-loop

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

RAJA execution policy types specify execution

```
RAJA::seq_exec

RAJA::omp_parallel_for_exec

RAJA::cuda_exec< BLOCK_SIZE, Async >
```

RAJA-style loop

```
using EXEC_POL = ...;

RAJA::forall< EXEC_POL >( RAJA::RangeSegment(0, N),
  [=] (int i) {
   y[i] = a * x[i] + y[i];
  }
);
```

Exercise #1: vector addition

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition.cpp** contains C-style sequential and OpenMP loops that add two vectors:

sequential

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

OpenMP

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

Exercise

Exercise #1: vector addition

- *Exercise*: *Implement RAJA sequential, OpenMP, and CUDA variants of the vector addition kernel.*
- *Code sections for you to fill in are indicated by comments in the exercise file* (RAJA/exercises/tutorial_halfday/ex1_vector-addition.cpp).
- *Run the code and check your results. The exercise file contains methods you can use to check your work and print results.*

See the policy section of the RAJA User Guide for a listing of RAJA loop execution policies.

Recall the basic RAJA loop transformation

C-style for-loop

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

RAJA execution policy types

```
RAJA::seq_exec

RAJA::omp_parallel_for_exec

RAJA::cuda_exec< BLOCK_SIZE, Async >
```

RAJA-style loop

```
using EXEC_POL = ...;

RAJA::forall< EXEC_POL >( RAJA::RangeSegment(0, N),
  [=] (int i) {
   y[i] = a * x[i] + y[i];
  }
);
```

Exercise #1 solution

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** contains a complete implementation of a solution for exercise #1
- Sequential:

C-style

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

RAJA-version

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

Note: The file also contains RAJA variants that use other "sequential" execution policies – **`simd_exec`**, **`loop_exec`**

Exercise #1 solution

- OpenMP (CPU):

C-style

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

RAJA-version

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

Exercise #1 solution

- CUDA:

C-style

Kernel definition

```
__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]; }
}
```

Kernel launch

```
addvec<<< grid_size, block_size >>>( c, a, b, N );
```

RAJA-version

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

Exercise #2: Reductions

Recall RAJA reduction objects

C-style dot product

```
double dot = 0.0;
for (int i = 0; i < N; ++i) {
  dot += a[i] * b[i];
}
```

RAJA reduction policy type must be compatible with execution policy type

```
RAJA::seq_reduce

RAJA::omp_reduce

RAJA::cuda_reduce
```

RAJA-style

```
RAJA::ReduceSum< REDUCE_POLICY, double> dot(0.0);

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

Exercise #2: approximate pi

- Recall a calculus identity:

$$\frac{\pi}{4} = \tan^{-1}(1) = \int_0^1 \frac{1}{1+x^2}\,dx$$

- The file **RAJA/exercises/tutorial_halfday /ex2_approx-pi.cpp** contains C-style sequential and OpenMP loops that use this formula to approximate pi using a *Riemann sum*.

C-style sequential

```
double pi = 0.0;
for (int i = 0; i < N; ++i) {
  double x = (i + 0.5) * dx;
  pi += dx / (1.0 + x * x);
}
pi *= 4.0;
```

C-style OpenMP

```
double pi = 0.0;
#pragma omp parallel for reduction(+:pi)
for (int i = 0; i < N; ++i) {
  double x = (i + 0.5) * dx;
  pi += dx / (1.0 + x * x);
}
pi *= 4.0;
```

Exercise

Exercise #2: approximate pi

- *Exercise: Implement RAJA variants for sequential, OpenMP, and CUDA using RAJA reductions.*
- *Code sections for you to fill in are indicated by comments in the exercise file* (RAJA/exercises/tutorial_halfday /ex2_approx-pi.cpp).
- *Run the code and check your results. The exercise file contains methods you can use to check your work and print results.*

See the policy section of the RAJA User Guide for a listing of RAJA loop execution and reduction policies.

Exercise

Recall RAJA reduction objects

C-style dot product

```
double dot = 0.0;
for (int i = 0; i < N; ++i) {
  dot += a[i] * b[i];
}
```

RAJA reduction policy types

```
RAJA::seq_reduce

RAJA::omp_reduce

RAJA::cuda_reduce
```

RAJA-style

```
RAJA::ReduceSum< REDUCE_POLICY, double> dot(0.0);

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

Exercise #2 solution

- The file **RAJA/exercises/tutorial_halfday/ex2_approx-pi_solution.cpp** contains a solution for the exercise. For example, the OpenMP solution looks like this:

C-style

```
double pi = 0.0;
#pragma omp parallel for \
reduction(+:pi)
for (int i = 0; i < N; ++i) {
  double x = (i + 0.5) * dx;
  pi += dx / (1.0 + x * x);
}
pi *= 4.0;
```

RAJA

```
RAJA::ReduceSum< RAJA::omp_reduce, double > pi_sum(0.0);
RAJA::forall< RAJA::seq_exec >(RAJA::RangeSegment(0, N),
  [=] (int i) {
    double x = (i + 0.5) * dx;
    pi_sum += dx / (1.0 + x * x);
  }
);
double pi = pi_sum.get() * 4.0;
```

The sequential and CUDA RAJA variants look the same, except for execution & reduction policies…

```
RAJA::seq_exec – RAJA::seq_reduce

RAJA::cuda_exec – RAJA::cuda_reduce
```

Exercise #3:
List Segments and IndexSets

Recall that Index Sets enable iteration space partitioning

```
using ListSegType = RAJA::TypedListSegment<int>;

int idx1[ ] = {1, 3, 4, 7};
ListSegType list1( idx, 4 );

int idx2[ ] = {10, 11, 14, 20, 22};
ListSegType list2( idx, 5 );

RAJA::TypedIndexSet< ListSegType > iset;

iset.push_back( list1 );
iset.push_back( list2 );
```

Iteration space is partitioned into 2 List Segments

{1, 3, 4, 7} (list1) & {10, 11, 14, 20, 22} (list2)

An Index Set can be passed to a RAJA execution template to run all Segments

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

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

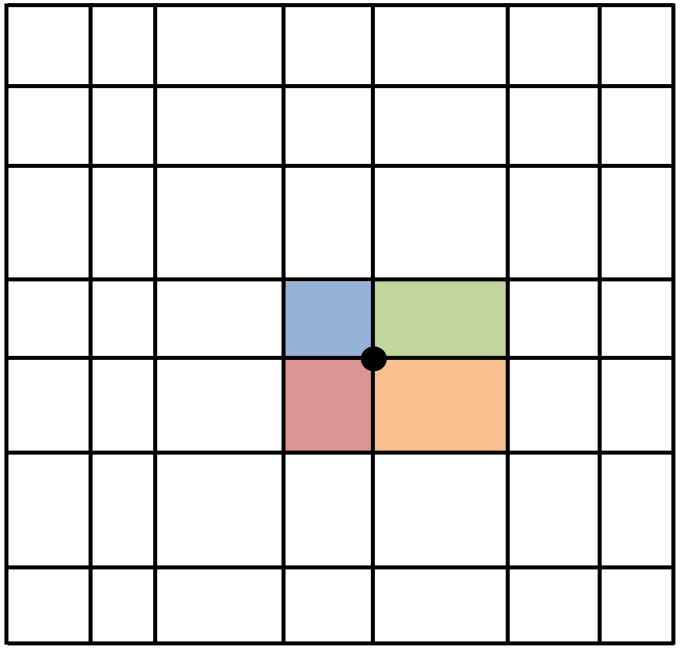
Index sets require a **two-level execution policy**:

- Outer iteration over segments ("…_segit")
- Inner segment execution

Exercise

IndexSets can help enable parallelism

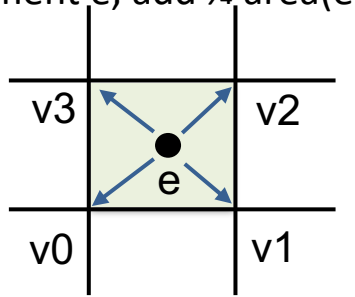
- Consider an irregularly-spaced 2D Cartesian mesh
- For each mesh vertex, we want to compute and store the average area of the 4 surrounding elements

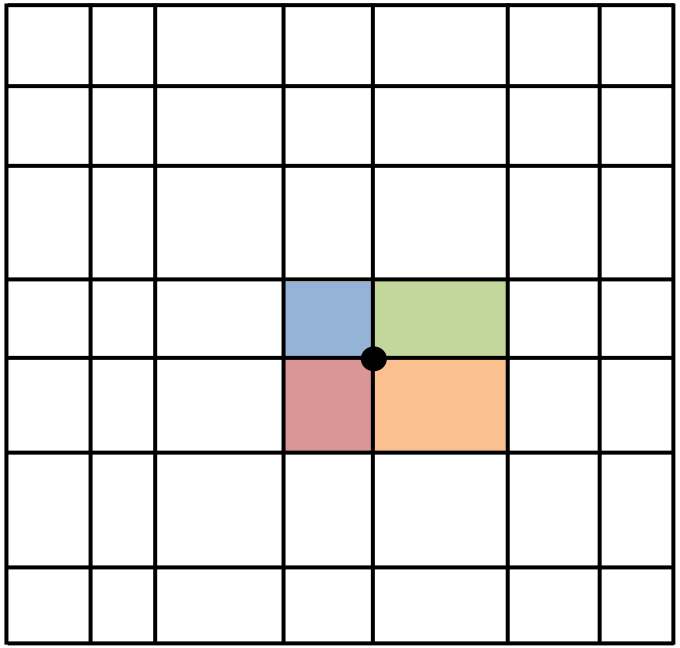


Exercise

IndexSets can help enable parallelism

- Consider an irregularly-spaced 2D Cartesian mesh
- For each mesh vertex, we want to compute and store the average area of the 4 surrounding elements
  - For each element e, add ¼ area(e) to area(vi), i = 0,…,3





Exercise

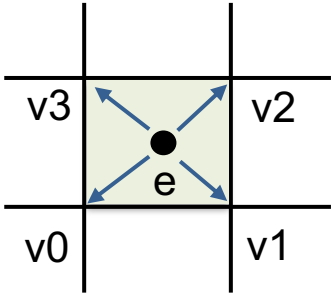
A C-style OpenMP code to compute vertex areas

```
#pragma omp parallel for
for (int ie = 0 ; ie < N_elem ; ++ie) {

  int* iv = &(e2v_map[4*ie]);

  areav[ iv[0] ] += areae[ie] / 4.0 ;
  areav[ iv[1] ] += areae[ie] / 4.0 ;
  areav[ iv[2] ] += areae[ie] / 4.0 ;
  areav[ iv[3] ] += areae[ie] / 4.0 ;

}
```



Will this code run correctly?

Exercise

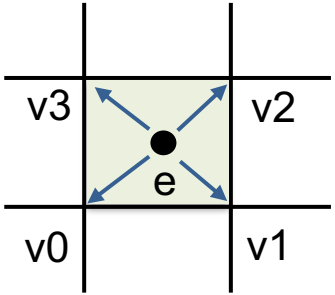
A C-style OpenMP code to compute vertex areas

```
#pragma omp parallel for
for (int ie = 0 ; ie < N_elem ; ++ie) {

  int* iv = &(e2v_map[4*ie]);

  areav[ iv[0] ] += areae[ie] / 4.0 ;
  areav[ iv[1] ] += areae[ie] / 4.0 ;
  areav[ iv[2] ] += areae[ie] / 4.0 ;
  areav[ iv[3] ] += areae[ie] / 4.0 ;

}
```



Will this code run correctly? Probably not. There is a data race at each vertex.

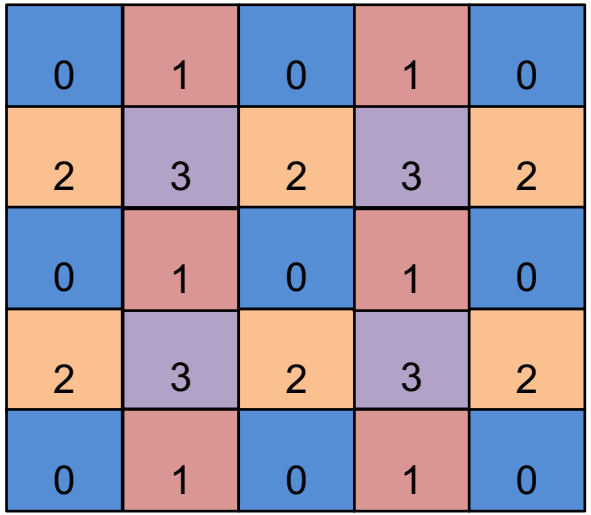
One solution: partition elements into four subsets (colors) and run each in parallel

Exercise

```
for (int icolor = 0; icolor < 4; ++icolor) {

  const std::vector<int>& ievec = idx[icolor];

  #pragma omp parallel for
  for (int i = 0; i < ievec.size(); ++i) {
    int ie = ievec[i];
    int* iv = &(e2v_map[4*ie]);
    areav[ iv[0] ] += areae[ie] / 4.0;
    areav[ iv[1] ] += areae[ie] / 4.0;
    areav[ iv[2] ] += areae[ie] / 4.0;
    areav[ iv[3] ] += areae[ie] / 4.0;
  }
}
```



No two elements with same color share a vertex

One approach: partition elements into four subsets (colors) and run each in parallel

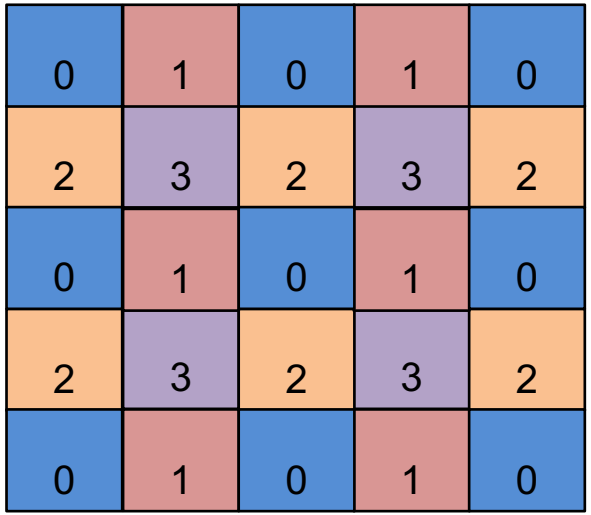
Exercise

```
for (int icolor = 0; icolor < 4; ++icolor) {

  const std::vector<int>& ievec = idx[icolor];

  #pragma omp parallel for
  for (int i = 0; i < ievec.size(); ++i) {
    int ie = ievec[i];
    int* iv = &(e2v_map[4*ie]);
    areav[ iv[0] ] += areae[ie] / 4.0;
    areav[ iv[1] ] += areae[ie] / 4.0;
    areav[ iv[2] ] += areae[ie] / 4.0;
    areav[ iv[3] ] += areae[ie] / 4.0;
  }
}
```

Will this code run correctly?



No two elements with same color share a vertex

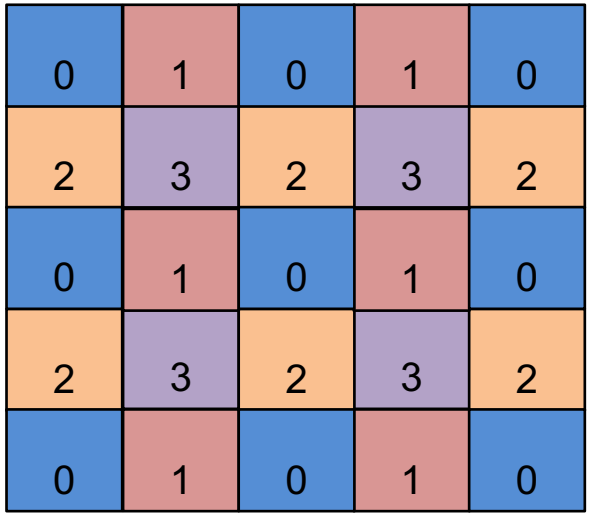
One approach: partition elements into four subsets (colors) and run each in parallel

Exercise

```
for (int icolor = 0; icolor < 4; ++icolor) {

  const std::vector<int>& ievec = idx[icolor];

  #pragma omp parallel for
  for (int i = 0; i < ievec.size(); ++i) {
    int ie = ievec[i];
    int* iv = &(e2v_map[4*ie]);
    areav[ iv[0] ] += areae[ie] / 4.0;
    areav[ iv[1] ] += areae[ie] / 4.0;
    areav[ iv[2] ] += areae[ie] / 4.0;
    areav[ iv[3] ] += areae[ie] / 4.0;
  }
}
```



No two elements with same color share a vertex

Will this code run correctly? Yes. The inner loop is data parallel

Exercise

Exercise #3: Mesh vertex area using an index set

- The file **RAJA/exercises /tutorial_halfday/ex3_colored-indexset.cpp** contains a C-style OpenMP variant like on the previous slide that uses arrays to enumerate elements of each color:

```
for (int icol = 0; icol < 4; ++icol) {

   const std::vector<int>& ievec = idx[icol];
   const int len = static_cast<int>(ievec.size());

   #pragma omp parallel for
   for (int i = 0; i < len; ++i) {
      int ie = ievec[i];
      int* iv = &(e2v_map[4*ie]);
      areav[ iv[0] ] += areae[ie] / 4.0 ;
      areav[ iv[1] ] += areae[ie] / 4.0 ;
      areav[ iv[2] ] += areae[ie] / 4.0 ;
      areav[ iv[3] ] += areae[ie] / 4.0 ;
   }

}
```

Exercise

Exercise #3: Mesh vertex area using an index set

- *Exercise: Implement RAJA OpenMP and CUDA variants of the vertex area calculation using a RAJA Index Set with 4 List Segments. Run in parallel over each segment!!*
- *You can use the integer arrays from the C-style OpenMP version to construct your List Segments for the Index Set.*
- *The exercise file* (RAJA/exercises /tutorial_halfday/ex3_colored-indexset.cpp) *contains RAJA Index Set execution policy types for each case and empty code sections for you to fill in.*
- *Run the code and check your results. The exercise file contains methods you can use to check your work and print results.*

Exercise #3 solution

- The file **RAJA/exercises/tutorial_halfday/ex3_colored-indexset_solution.cpp** contains a complete implementation of a solution for exercise #3.

- The code for generating the RAJA IndexSet (used in all cases) look likes the following:

```
using SegmentType = RAJA::TypedListSegment<int>;

RAJA::TypedIndexSet<SegmentType> colorset;

colorset.push_back( SegmentType(&idx[0][0], idx[0].size()) );
colorset.push_back( SegmentType(&idx[1][0], idx[1].size()) );
colorset.push_back( SegmentType(&idx[2][0], idx[2].size()) );
colorset.push_back( SegmentType(&idx[3][0], idx[3].size()) );
```

- Here, 'idx' is an array of arrays that define the element indices for each of the four colors. These arrays are used in the non-RAJA OpenMP variant shown on an earlier slide.

Exercise #3 solution

- The kernel code for the RAJA OpenMP vertex area implementation looks like this:

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

RAJA::forall< EXEC_POL >(colorset, [=] (int ie) {
    int* iv = &(e2v_map[4*ie]);
    areav[ iv[0] ] += areae[ie] / 4.0 ;
    areav[ iv[1] ] += areae[ie] / 4.0 ;
    areav[ iv[2] ] += areae[ie] / 4.0 ;
    areav[ iv[3] ] += areae[ie] / 4.0 ;
});
```

Note that indirection does not appear inside the kernel for element indexing.

For the RAJA CUDA variant, the inner OpenMP segment execution policy is replaced with a CUDA policy

```
RAJA::cuda_exec< CUDA_BLOCK_SIZE >
```

Exercise #4:
Atomic operations

Recall exercise #2

- We approximated π using a Riemann sum for the formula:

$$\frac{\pi}{4} = \tan^{-1}(1) = \int_0^1 \frac{1}{1+x^2}\,dx$$

- We used a RAJA reduction to accumulate the Riemann sum in parallel
- We can also use an atomic operation to prevent multiple threads from attempting to write to the sum variable at the same time

RAJA OpenMP atomic 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;
```

Remember that the atomic policy must be compatible with the loop execution policy (just like reductions).

Exercise

Exercise #4: atomic histogram

- The file **RAJA/exercises/tutorial_halfday/ex4_atomic-histogram.cpp** describes a histogram calculation:
  - Given an integer array of length N with entries in the set {0, 1, 2, …, M-1}, where M < N. Fill in entries of an array of length M so that the i-th array entry is the number of occurrences of the value ‘i’ in the original array
- The C-style OpenMP implementation looks like this:

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  #pragma omp atomic
  hist[ array[i] ]++;
}
```

Exercise

Exercise #4: atomic histogram

- *Exercise: Implement RAJA variants of the histogram calculation for OpenMP and CUDA using RAJA atomic operations.*
- *The exercise file* (RAJA/exercises/tutorial_halfday/ex4_atomic-histogram.cpp) *asks for two versions of each, one using the programming model-specific RAJA atomic policy, and one using the 'auto' atomic policy.*
- *The exercise file contains empty code sections for you to fill in and methods you can use to check your work and print results.*
- *Run the code and check your results. The file contains methods you can use to check your work and print results.*

Exercise #4 solution

- The file **RAJA/exercises/tutorial_halfday/ex4_atomic-histogram_solution.cpp** contains a complete implementation of the solution for exercise #4. For example, for OpenMP:

C-style

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  #pragma omp atomic
  hist[ array[i] ]++;
}
```

The RAJA CUDA version is similar but uses the **cuda_atomic** policy. Alternatively, either version can use the RAJA **auto_atomic** policy.

RAJA Version

```
RAJA::forall< RAJA::omp_parallel_for_exec >(RAJA::RangeSegment(0, N),
[=] (int i) {
  RAJA::atomicAdd< RAJA::omp_atomic >(&hist[array[i]], 1);
});
```

Exercise #5:
Scan operations

Recall RAJA scan operations

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

For example,

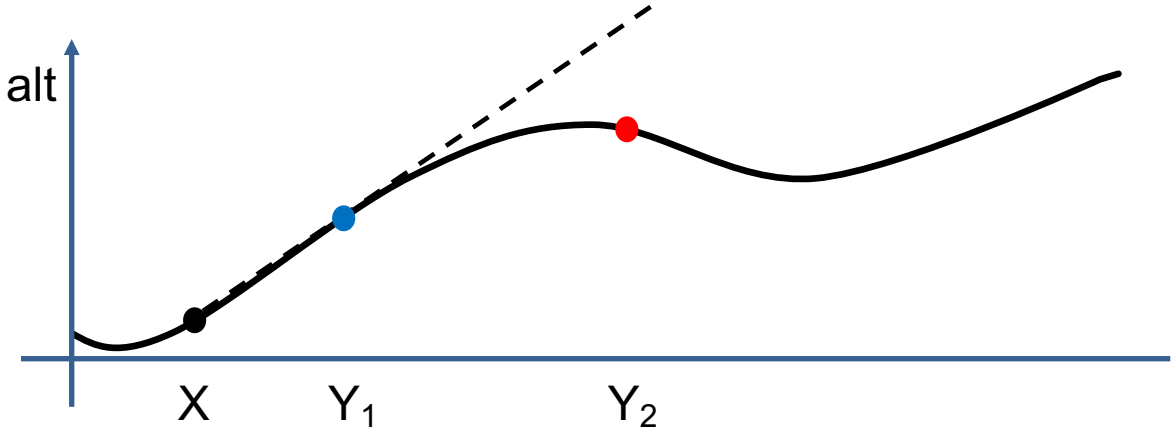
Input array : 8 -1 2 9 10 -3 4 1 6 7

Output array : 8 -1 -1 -1 -1 -3 -3 -3 -3 -3

If no operator is given, “plus” is the default (prefix-sum).

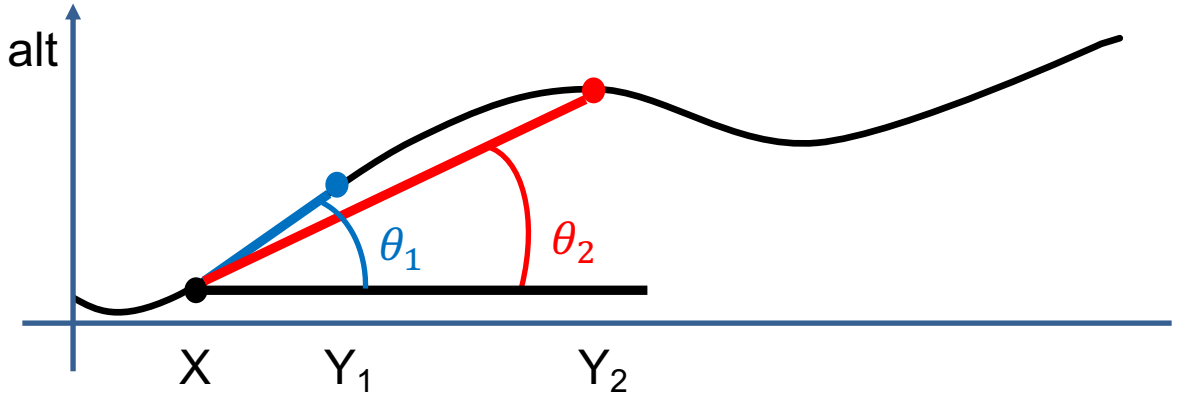
Exercise #5: the line-of-sight problem

- The *line-of-sight* problem: given an observation point at X on a terrain map, and a set of points along a ray starting at X, find which points on the terrain are visible from X.
- For example, the blue point at Y_1 is visible from the black point at X, but the red point at Y_2 is not



Exercise #5: the line-of-sight problem

- A point at Y on the surface is visible from the point at X if and only if no other point on the terrain between the points X and Y has a greater vertical angle from X than Y.



Although the point at Y_2 has a higher altitude than the point at Y_1, it has a smaller vertical angle; i.e., $\theta_2 < \theta_1$ Thus, the point at Y_2 cannot be seen from the point at X.

Exercise #5: the line-of-sight problem

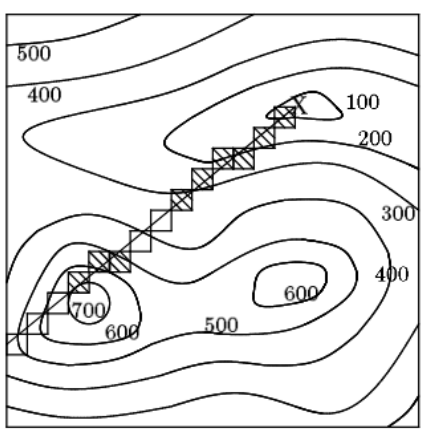
- Algorithm for determining which terrain points at Y_i are visible from X:

- Let ‘altX’ be the altitude at point X and let ‘alt’ be a vector such that alt[i] is the altitude at point Y_i.

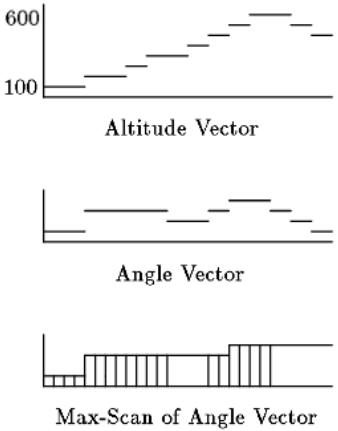
- Let ‘dist’ be a vector such that dist[i] is the horizontal distance between point X and point Y_i.

- Compute an angle vector ‘ang’ such that ang[i] is the vertical angle from point X to point Y_i :
  - ang[i] = $\tan^{-1}$ ((alt[i] – altX)/dist[i])

- A *max scan* on the angle vector gives us a vector ‘ang_max’. Then, we can use it to see if Y_i is visible:
  - If ang[i] >= ang_max[i], then Y_i is visible from X, else Y_i is not visible



Altitude Map



Ray Vectors

Image reference:
“Prefix Sums and Their Applications” by Guy E. Blelloch (https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf)

Exercise #5: the line of sight problem

- The file **RAJA/exercises/ex5_line-of-sight.cpp** contains a C-style sequential implementation of the line-of-sight algorithm ('ang' array has already been filled in for you):

```
ang_max[0] = ang[0];
for (int i = 1; i < N; ++i) {
  ang_max[i] = std::max(ang[i], ang_max[i-1]);
}

for (int i = 0; i < N; ++i) {
  if ( ang[i] >= ang_max[i] ) {
    visible[i] = 1;
  } else {
    visible[i] = 0;
  }
}
```

Exercise #5: the line of sight problem

- *Exercise: Implement Sequential, OpenMP, and CUDA variants of the algorithm using a RAJA scan operation to compute the max angle vector and a RAJA::forall kernel to determine which points are visible.*

- *The exercise file* (RAJA/exercises/ex5_line-of-sight.cpp) *contains empty code sections for you to fill.*

- *The file also contains methods you can use to check your work and print results.*

Recall that execution policies for RAJA scan methods are the same as ‘forall’ policies.

Exercise #5 solution

- The file **RAJA/exercises/tutorial_halfday/ex5_line-of-sight_solution.cpp** contains a complete implementation of the solution to exercise #5.

- The solution looks like this, where the appropriate execution policy is used:

```
RAJA::inclusive_scan< EXEC_POL >(ang, ang+N, ang_max,
                                 RAJA::operators::maximum<double>{});

RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N), [=] (int i) {
    if ( ang[i] >= ang_max[i] ) {
       visible[i] = 1;
    } else {
       visible[i] = 0;
    }
})
```

Note that we use an ‘inclusive’ (max) scan. Is it clear why we do this?

Exercise #6:
Views and Layouts

Recall that an offset layout applies an offset to indices

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

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 < 6; ++i) {
  CView(i) = ...;
}
```

A 1-d View with index offset and extent 11 [-5, 5].

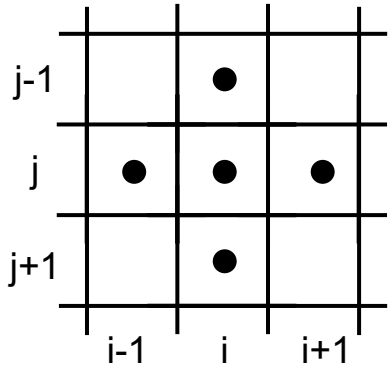
-5 will be subtracted from each loop index to access array data.

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

Exercise #6: 5-point stencil

- Consider a simple “five-point stencil” operation on a 2-dimensional cartesian mesh

$$A_{i,j} = B_{i,j} + B_{i-1,j} + B_{i+1,j} + B_{i,j-1} + B_{i,j+1}$$



- Suppose the “A” array has an entry $A_{i,j}$ for each element on the mesh interior:

$$(i,j) \in \{0,\ldots,N\} \times \{0,\ldots M\}$$

and the “B” array has an entry $B_{i,j}$ for each element on the mesh interior plus a “halo” layer 1 element wide around the interior:

$$(i,j) \in \{-1,\ldots,N+1\} \times \{-1,\ldots,M+1\}$$

Exercise #6: 5-point stencil

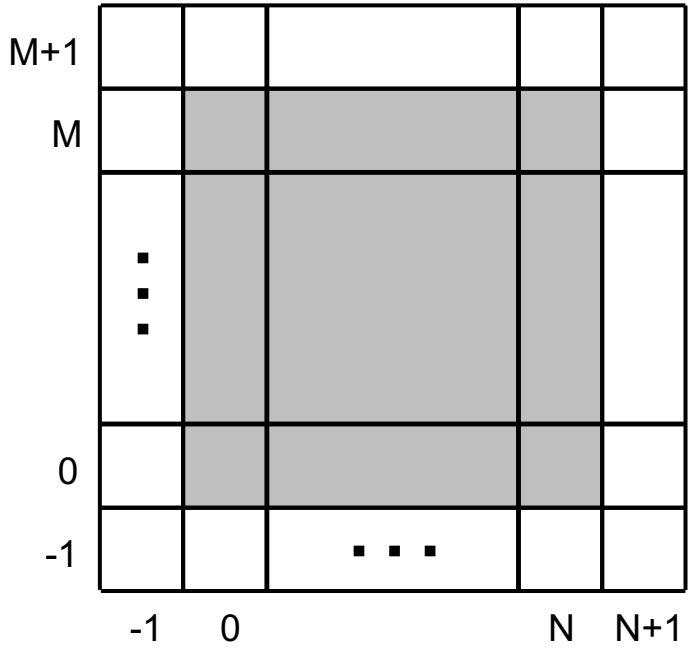
- That is, B has a value for each element on the mesh to the right and A has a value for each element in the grey interior region

- We want to write the stencil computation as a nested loop (i, j) using RAJA Views to write the loop body "naturally" as:

Aview(i, j) = Bview(i, j) + Bview(i-1, j) + Bview(i+1, j) +
Bview(i, j-1) + Bview(i, j+1)

- That is, so it looks like the math formula:

$$A_{i,j} = B_{i,j} + B_{i-1,j} + B_{i+1,j} + B_{i,j-1} + B_{i,j+1} \quad \text{where} \quad (i,j) \in \{0,\dots,N\} \times \{0,\dots M\}$$



Exercise #6: 5-point stencil

- The file **RAJA/exercises/ex6_stencil-offset-layout.cpp** contains two C-style sequential implementations of the 5-point stencil computation (note: manual index offsets are used):
  - Part A: assume the column index (j-loop) is stride-1

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

    int idx_out = j + Nr_int * i;                // 'A' index
    int idx_in = (j + 1) + Nr_tot * (i + 1);     // 'B' index

    A[idx_out] = B[idx_in] +                                    // C
                 B[idx_in - Nr_tot] + B[idx_in + Nr_tot] +      // W, E
                 B[idx_in - 1] + B[idx_in + 1];                 // S, N

  }
}
```

Exercise #6: 5-point stencil

- The file **RAJA/exercises/ex6_stencil-offset-layout.cpp** contains two C-style sequential implementations of the 5-point stencil computation (note: manual index offsets are used):
  - Part B: assume the row index (i-loop) is stride-1

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

    int idx_out = i + Nc_int * j;                 // 'A' index
    int idx_in = (i + 1) + Nc_tot * (j + 1);      // 'B' index

    A[idx_out] = B[idx_in] +                                    // C
                 B[idx_in - Nc_tot] + B[idx_in + Nc_tot] +      // S, N
                 B[idx_in - 1] + B[idx_in + 1];                 // W, E

  }
}
```

C-style implementations for Parts A and B differ only in the nested loop order and the index offset arithmetic.

Exercise

Exercise #6: 5-point stencil

- *Exercise: Implement sequential variants of parts A and B using RAJA Views.*
- *The goal of this exercise is for you to learn the mechanics of building and using RAJA Layouts and Views so* ***RAJA execution methods are not used – use the same loops as the C-style variants.***
- *The exercise file* (RAJA/exercises/ex6_stencil-offset-layout.cpp) *contains empty code sections to fill in.*
- The goal is to use RAJA Views to make the loop bodies the same in each case; like this:

$$Aview(i, j) = Bview(i, j) + Bview(i-1, j) + Bview(i+1, j) + Bview(i, j-1) + Bview(i, j+1)$$

- *The exercise file contains methods you can use to check your work and print results.*

Exercise #6: 5-point stencil

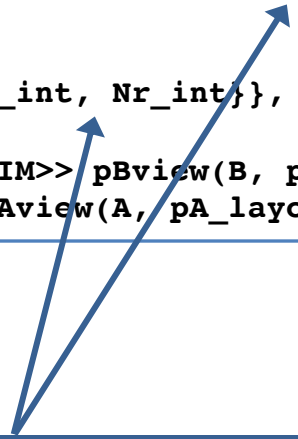
- File **RAJA/exercises/ex6_stencil-offset-layout_solution.cpp** contains a complete implementation of the solution to the exercise. Here's how the views are defined for part B:

```
std::array<RAJA::idx_t, DIM> perm {{1, 0}};  // index stride permutation

RAJA::OffsetLayout<DIM> pB_layout =
    RAJA::make_permuted_offset_layout( {{-1, -1}}, {{Nc_int, Nr_int}}, perm );

RAJA::Layout<DIM> pA_layout =
    RAJA::make_permuted_layout( {{Nc_int, Nr_int}}, perm );

RAJA::View<int, RAJA::OffsetLayout<DIM>> pBview(B, pB_layout);
RAJA::View<int, RAJA::Layout<DIM>> pAview(A, pA_layout);
```



Note: the offset layout args express an **index range**, while the other layout args express **index extents**.

Exercise #6 solution

- The loops for the stencil computation for part B look like this:

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

      pAview(i, j) = pBview(i, j) +                              // C
                     pBview(i - 1, j) + pBview(i + 1, j) +       // W, E
                     pBview(i, j - 1) + pBview(i, j + 1);        // S, N

   }
}
```

Part A is similar, except that the Views are not permuted and the loop nest order is reversed. Also, the names of the Views are different (since the code for both variants is in the same scope). **The loop kernel is the same in both cases because you are using RAJA Views.**

Nested loop reordering

Recall that each loop level has an iteration space and an execution policy when using the RAJA::kernel API…

```
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.

… and that to change nested loop order, you change the execution policy, not the kernel code

```
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.

Exercise #7: Nested loop reordering

- The file **RAJA/exercises/ex7_nested-loop-reorder.cpp** contains a 3-level C-style loop nest and a RAJA::kernel variant of the same

C-style

```
for (int k = 2; k < 4; ++k) {
  for (int j = 1; j < 3; ++j) {
    for (int i = 0; i < 2; ++i) {
      printf(" (%d, %d, %d) \n", i, j, k);
    }
  }
}
```

RAJA::kernel variant

```
using KJI_EXECPOL = KernelPolicy<
          statement::For<2, seq_exec,      // k
            statement::For<1, seq_exec,    // j
              statement::For<0, seq_exec, // i
                statement::Lambda<0>
              >
            >
          >
        >;

RAJA::kernel<EXECPOL>(
  RAJA::make_tuple(IRange, JRange, KRange),
  [=] (IIDX i, JIDX j, KIDX k) {
    printf("(%d, %d, %d) \n",
           (int)(*i), (int)(*j), (int)(*k));
  }
);
```

Note: order of ranges in the tuple must match the lambda argument list.

Exercise #7: Nested loop reordering

- The exercise uses RAJA's strongly-typed indices. We provide these and the nested loop range segments for you:

```
RAJA_INDEX_VALUE(KIDX, "KIDX");
RAJA_INDEX_VALUE(JIDX, "JIDX");
RAJA_INDEX_VALUE(IIDX, "IIDX");

RAJA::TypedRangeSegment<KIDX> KRange(2, 4);
RAJA::TypedRangeSegment<JIDX> JRange(1, 3);
RAJA::TypedRangeSegment<IIDX> IRange(0, 2);
```

```
RAJA::kernel<EXECPOL>( RAJA::make_tuple(IRange, JRange, KRange),
  [=] (IIDX i, JIDX j, KIDX k) {
     printf( " (%d, %d, %d) \n", (int)(*i), (int)(*j), (int)(*k));
  });
```

Exercise #7: Nested loop reordering

- *Exercise: Implement two other sequential variants by defining RAJA::KernelPolicy types that permute the loop nest ordering:*
  - *One: j-loop is the outer loop, k-loop is the inner loop, and the i-loop in the middle*
  - *The other: i-loop as the outer loop, j-loop as the inner inner, and the k-loop in the middle*

- Only the policies should differ between the two. The kernel should look the same for any ordering of the loop nest:

```
RAJA::kernel<EXECPOL>( RAJA::make_tuple(IRange, JRange, KRange),
  [=] (IIDX i, JIDX j, KIDX k) {
     printf( " (%d, %d, %d) \n", (int)(*i), (int)(*j), (int)(*k));
  });
```

The order of ‘For’ statements in the execution policy types determines the loop nest order.

Exercise #7 solution

- The file **RAJA/exercises/ex7_nested-loop-reorder_solution.cpp** contains a complete implementation of the exercise solution

- Ordering: j outer, i middle, k inner

```
using JIK_EXECPOL = KernelPolicy<
          statement::For<1, seq_exec,      // j
            statement::For<0, seq_exec,    // i
              statement::For<2, seq_exec, // k
                statement::Lambda<0>
              ...
```

```
for (j = 1; j < 3; ++j) {
  for (i = 0; i < 2; ++i) {
    for (k = 2; k < 4; ++k) {
      ...
```

- Ordering: i outer, k middle, j inner

```
using IKJ_EXECPOL = KernelPolicy<
          statement::For<0, seq_exec,      // i
            statement::For<2, seq_exec,    // k
              statement::For<1, seq_exec, // j
                statement::Lambda<0>
              ...
```

```
for (i = 0; i < 2; ++i) {
  for (k = 2; k < 4; ++k) {
    for (j = 1; j < 3; ++j) {
      ...
```

Loop Tiling

Recall the C-style matrix transpose operation with loop tiling we discussed earlier

$A^T(c, r) = A(r, c)$, 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 col = bc * TILE_SZ + tc;  // global column index
        int row = br * TILE_SZ + tr;  // global row index

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

      }
    }
  }
}
```

Notes:
- RAJA views for A and At are used here.
- Row, col bounds checks are needed in general.

Recall that RAJA provides tiling statements to simplify tiled loop index calculations

```
using namespace RAJA;

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

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

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

          >
        >
      >
    >
  >;
```

Nested loop constructs inside tile statements are the same as in the non-tiled case.

Note that global indices are passed to the lambda body automatically.

Exercise

Exercise #8: Tiled matrix transpose

- File **RAJA/exercises/tutorial_halfday/ex8_tiled-matrix-tranpose.cpp** contains a C-style sequential implementation of a tiled matrix transpose operation as shown on an earlier slide.
- *Exercise*: *Implement RAJA kernel variants of the matrix transpose operation for sequential, OpenMP, and CUDA execution.*
- *Partial execution policies are provided that include the tiling statements. Your task is to fill in the missing policy statements and loop kernels as indicated in the file.*
- *The exercise file contains methods you can use to check your work and print results.*
- *Note that there are two OpenMP variants included: one that parallelizes the top inner tile loop (you will do this one) and one that collapses the two inner tile loops (this one is done for you).*

Exercise #8 Solution

- The file **RAJA/exercises/tutorial_halfday/ex8_tiled-matrix-transpose_solution.cpp** contains a complete implementation of the solution to exercise #8.
- The code looks like the following for each case:

Different 'inner loop' execution policy types are used in each case. The kernel code is the same.

```
using KERNEL_POL = KernelPolicy<
      statement::Tile<1, statement::tile_fixed<TILE_SZ>, OUT_TILE_POL1,
        statement::Tile<0, statement::tile_fixed<TILE_SZ>, OUT_TILE_POL2,
          statement::For<1, IN_TILE_POL1,
            statement::For<0, IN_TILE_POL2,
              statement::Lambda<0>
            >
          >
        >
      >
    >;

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

You fill in these parts.

Exercise #8 Solution

- The sequential kernel defines each loop execution policy type to be **`seq_exec`**
- Here's the policy for the kernel that applies OpenMP to the top inner tile loop:

```
using KERNEL_POL = KernelPolicy<
      statement::Tile<1, statement::tile_fixed<TILE_SZ>, seq_exec,
        statement::Tile<0, statement::tile_fixed<TILE_SZ>, seq_exec,
          statement::For<1, omp_parallel_for_exec,
            statement::For<0, seq_exec,
              statement::Lambda<0>
            >
          >
        >
      >
    >;
```

Exercise #8 Solution

- Here's the policy for the CUDA kernel:

```
using KERNEL_POL = KernelPolicy<
      statement::Tile<1, statement::tile_fixed<TILE_SZ>, cuda_block_y_loop,
        statement::Tile<0, statement::tile_fixed<TILE_SZ>, cuda_block_x_loop,
          statement::For<1, cuda_thread_y_direct,
            statement::For<0, cuda_thread_x_direct,
              statement::Lambda<0>
            >
          >
        >
      >
    >;
```

Loop tiling and local data

Recall RAJA::kernel_param: it takes a tuple for thread/kernel-local data

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

  RAJA::make_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(row, k) * B(k, col);
  },

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

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

The kernel uses three lambdas and the execution policy indicates where they are invoked

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

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


      >
    >
  >;
```

Back to the matrix transpose example with loop tiling…

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

    int Tile[TILE_SZ][TILE_SZ];    // declare local tile array

    for (int tr = 0; tr < TILE_SZ; ++tr) {     // Read a tile of 'A'
      for (int tc = 0; tc < TILE_SZ; ++tc) {
        int col = bc * TILE_SZ + tc;  // global column index
        int row = br * TILE_SZ + tr;  // global row index
        if (row < N_r && col < N_c) { Tile[tr][tc] = A(row, col); }
      }
    }

    // Write tile to 'At' in another loop nest over the tile
  }
}
```

C-style 'tiled' loop nest

As in exercise #8, global matrix indices are manually calculated here.
Here, we also need to use the local tile indices to read/write data from/to local tile array.

Kernel policy statements allow you to access the local tile indices within a lambda expression

```
using namespace RAJA;
using TILE_MEM = LocalArray<int, Perm<0, 1>, SizeList<TILE_SZ, TILE_SZ>>;

TILE_MEM TileArray;

using EXEC_POL = KernelPolicy<

  // statements for outer loops over tiles

    statement::InitLocalMem<tile_mem_policy, ParamList< # >,

    statement::ForICount<1, statement::Param<1>, TILE_POL_1,
      statement::ForICount<0, statement::Param<0>, TILE_POL_0,
        statement::Lambda<0>   // lambda to read tile of 'A' into local array
      >
    >

   // Write tile to 'At' in another 'ForICount' loop nest over tile

>;
```

'ForICount' statements generate local tile indices that can be used in lambda expressions

- First integer arg indicates iteration space tuple entry (as in regular 'For'
- 'Param' statement indicates local data tuple entry for local index

The local array can be accessed in any lambda in the kernel

Local Data

```
RAJA::kernel_param<EXEC_POL>(
  RAJA::make_tuple(RAJA::RangeSegment(0, N_c), RAJA::RangeSegment(0, N_r)),

  RAJA::make_tuple((int)0, (int)0, TileArray),

  [=](int col, int row, int tc, int tr, TILE_MEM& TileArray) {
    TileArray(ty, tx) = Aview(row, col);
  },

  [=](int col, int row, int tc, int tr, TILE_MEM TileArray) {
    Atview(col, row) = TileArray(ty, tx);
  }
);
```

Local indices tc, tr are first two entries in param tuple

Lambda args:
- Global indices (from ranges)
- Local tile indices
- LocalArray for tile data

Exercise #9: Tiled matrix transpose with local array

- File **RAJA/exercises/tutorial_halfday/ex9_tiled-matrix-transpose-local-array.cpp** contains a C-style sequential implementation of a tiled matrix transpose operation:

```
for (int by = 0; by < outer_Dimr; ++by) {      // outer loops over tiles
  for (int bx = 0; bx < outer_Dimc; ++bx) {

    int Tile[TILE_SZ][TILE_SZ];   // stack-allocated local array

    for (int trow = 0; trow < TILE_SZ; ++trow) {   // inner loops in a tile
      for (int tcol = 0; tcol < TILE_SZ; ++tcol) {

        int col = bx * TILE_SZ + tcol;  // matrix column index
        int row = by * TILE_SZ + trow;  // matrix row index

        if (row < N_r && col < N_c) { Tile[trow][tcol] = Aview(row, col); }

      }
    }

    // etc.
```

The first part (shown here) copies a tile of A into the local array. The second part (not shown here) copies the transposed tile data into At.

Exercise #9: Tiled matrix transpose with local array

- *Exercise: Implement RAJA kernel variants of the matrix transpose operation for sequential, OpenMP, and CUDA execution using a local tile array. Partial execution policies are provided that have the tiling statements filled in. Your task is to fill in the missing statements as indicated.*

- *The exercise file contains methods you can use to check your work and print results.*

Notes:

- RAJA Layouts, Views, row and column segments are provided for you and are identical to those used in exercise #8.
- The RAJA::LocalArray object is created for you outside of the kernels. Its memory is not allocated until the 'InitiLocalMem' statement is encountered in each kernel policy.
- Each kernel uses **two lambda expressions** (one to write data into the local tile array, and one to read from it).
- 'ForICount' statements generate local tile indices passed to lambdas in kernel.

Exercise #9: Tiled matrix transpose with local array

- The kernel code looks like this in each case (note two lambdas!):

```
kernel_param<KERNEL_POL>( make_tuple(col_range, row_range),

                          make_tuple((int)0, (int)0, RAJA_Tile),

  [=] (int col, int row, int tcol, int trow, TILE_MEM& RAJA_Tile) {
    RAJA_Tile(trow, tcol) = Aview(row, col);
  },

  [=] (int col, int row, int tcol, int trow, TILE_MEM& RAJA_Tile) {
    Atview(col, row) = RAJA_Tile(trow, tcol);
  }

);
```

You are asked to fill in one of these lambdas.

Note that the global row-col indices are the first two lambda args. The other lambda args are generated from the items in the parameter tuple passed to the kernel_param method.

Exercise #9 Solution

(See file RAJA/exercises/tutorial_halfday/ex9_matrix-transpose-local-array_solution.cpp)

- The policy code for each case looks like this:

Different execution policy types are used in each kernel policy.

You fill in these parts.

```
using KERNEL_POL = KernelPolicy<
      Tile<1, tile_fixed<TILE_SZ>, OUT_TILE_POL1,
        Tile<0, tile_fixed<TILE_SZ>, OUT_TILE_POL2,

          InitLocalMem< MEM_POL >, ParamList<2>,

          ForICount<1, Param<0>, IN_TILE_POL1,
            ForICount<0, Param<1>, IN_TILE_POL2,
              Lambda<0>
            >
          >
          ForICount<0, Param<1>, IN_TILE_POL3,
            ForICount<1, Param<0>, IN_TILE_POL4,
              Lambda<1>
            >
          >
        >
      >
    >;
```

'ForICount' statements generate local tile indices passed to lambdas in kernel.

'Param' statements identify position of local index in param tuple.

'ParamList' statement indicates position of local array in param tuple.

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)

Related software is also available

- 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

More related software…

- 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

Wrap-up

Again, we would appreciate your feedback…

- If you have comments, questions, suggestions, etc., please talk to one of us
- Or contact us via our team email list: raja-dev@llnl.gov

Thank you for your attention and participation

Questions?