A Tutorial Introduction to RAJA

ECP Annual Meeting

February 3-7, 2020

EXASCALE COMPUTING PROJECT

Rich Hornung (hornung1@llnl.gov)
Arturo Vargas (vargas45@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

Welcome to the RAJA tutorial

- Today, we will describe RAJA and how it enables performance portability
- We will also present some background material that may help you think about key issues when developing parallel applications
- We will present examples that show you how to use RAJA
  - In the interest of time, we will walk through exercises as a group rather than have you work through them here individually
- Our objective for today is to teach you enough to start using RAJA 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
  - Join our Google Group (linked on RAJA GitHub project home page)
  - Or send a message to our project email list: raja-dev@llnl.gov
- We appreciate specific, concrete feedback that helps us improve RAJA and this tutorial

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 design goals target usability and developer productivity

- We want applications to maintain **single-source kernels** (as much as possible)
- In addition, we want RAJA to…
  - Be **easy to understand and use** for app developers (esp. those who are not CS experts)
  - Allow **incremental and selective adoption**
  - **Not force major disruption** to application source code
  - Promote flexible algorithm implementations via **clean encapsulation**
  - Make it **easy to parameterize execution** via type aliases
  - Enable **systematic performance tuning**

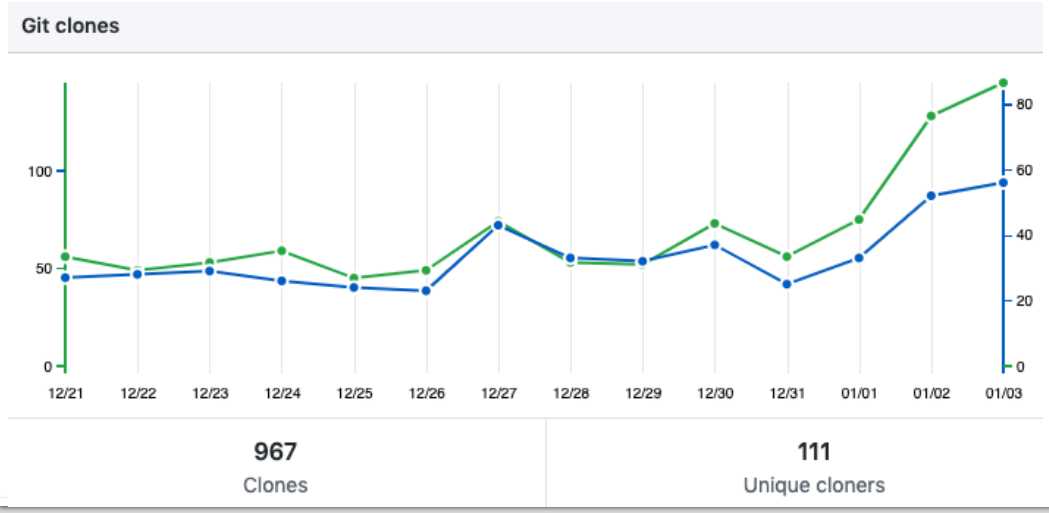
These goals have been affirmed by production application teams using RAJA.

RAJA is an open source project (https://github.com/LLNL/RAJA)

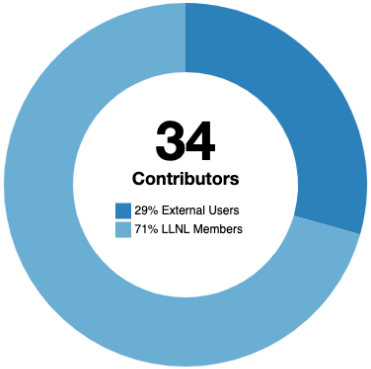
RAJA

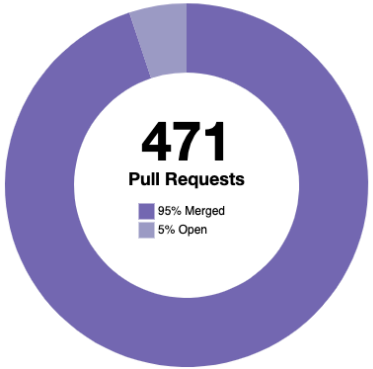
LLNL | C++ | BSD-3-Clause

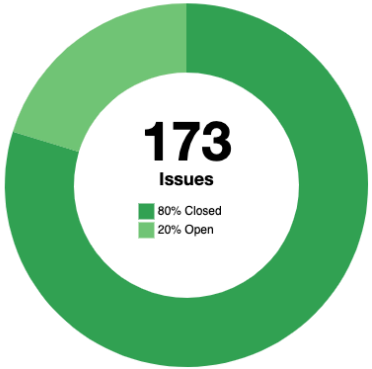
GitHub Page Stargazers : 161 Forks : 47



GitHub activity
1/19 – 1/20

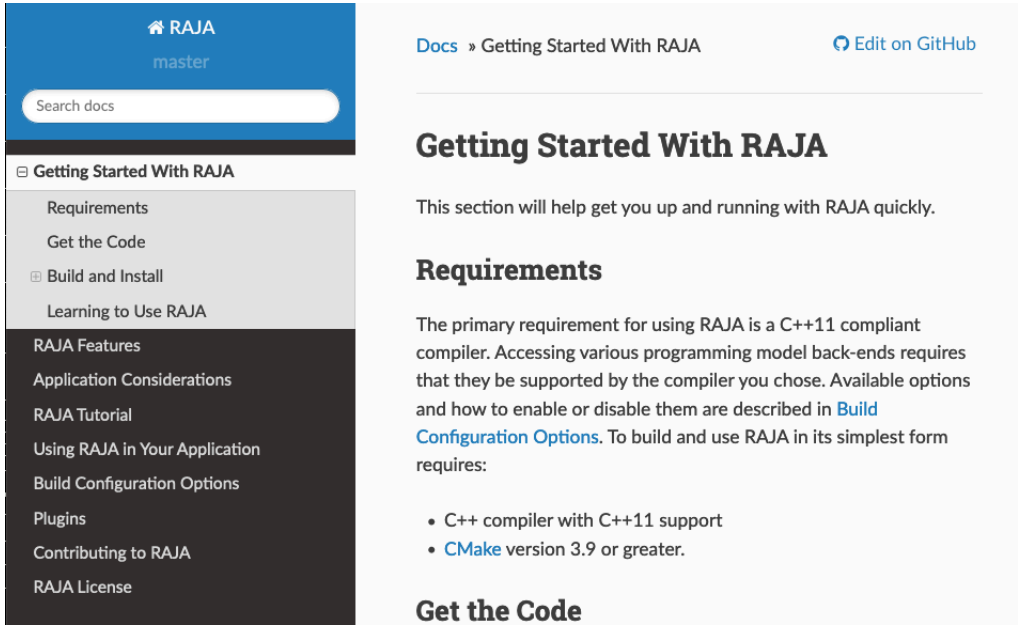






We also maintain other related projects…

- **RAJA User Guide:** getting started info, details about features and usage, etc. (readthedocs.org/projects/raja)
- **RAJA Project Template:** shows how to use RAJA and BLT in an application that uses CMake (https://github.com/LLNL/RAJA-project-template)
- **RAJA Performance Suite:** loop kernels for assessing compilers and RAJA performance. Used by us, vendors, for DOE platform procurements, etc. (https://github.com/LLNL/RAJAPerf)
- **CHAI:** array abstraction library that automatically migrates data as needed based on RAJA execution contexts (https://github.com/LLNL/CHAI)



All of these are linked on the RAJA GitHub project page.

We will cover various topics today

- RAJA usage considerations (C++ templates, lambdas, memory management, etc.)
- Generally useful information for reasoning about parallel algorithms
- RAJA features:
  - Simple loops
  - Reductions
  - Iteration spaces
  - Atomic operations
  - Scan operations
  - Data layouts and views
  - Complex loop kernels, including some advanced topics

Let's start simple…

Simple loop execution

Consider a typical 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 that 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 encapsulates loop execution details

C-style for-loop

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

Typically, definitions like these go in header files.

```
using EXEC_POL = ...;

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

RAJA-style loop

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

By changing the “execution policy” and “iteration space”, you change the way the loop runs.

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

C-style for-loop

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

RAJA-style loop

```
using EXEC_POL = ...;

RAJA::RangeSegment it_space(0, N);

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

Same loop body.

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)

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


```
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>
```

```
RAJA::omp_target_parallel_for_exec<MAX_THREADS_PER_TEAM>
```

```
RAJA::tbb_for_exec
```

A sampling of RAJA loop execution policy types.

RAJA provides a variety of execution policy types...

- Sequential (forces strictly sequential execution)
- “Loop” (lets compiler decide which optimizations to apply)
- SIMD (applies vectorization pragmas)
- OpenMP multithreading (CPU)
- TBB** (Intel Threading Building Blocks)
- CUDA (NVIDIA GPUs)
- OpenMP target** (available target device; e.g., GPU)
- HIP** (AMD GPUs)

Some execution back-ends are **works-in-progress.

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

Before we continue, let's discuss a few RAJA usage considerations

RAJA makes heavy use of C++ templates

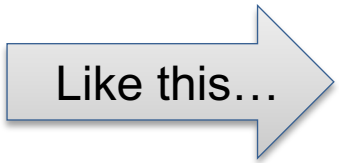
```
template <typename ExecPol,
          typename IdxType,
          typename LoopBody>
forall(IdxType&& idx, LoopBody&& body) {
   ...
}
```

- Templates allow one to write *generic* code and have the *compiler generate* a specific implementation for each set of template parameter types specified
- Here, “ExecPol”, “IdxType”, “LoopBody” are C++ types you provide at compile-time

RAJA makes heavy use of C++ templates

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

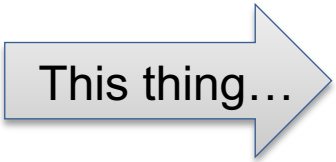
- "ExecPol", "IdxType", "LoopBody" are C++ types you specify




```
forall< seq_exec >( RangeSegment(0, N), ...
   // loop body
);
```

- Note: "IdxType" and "LoopBody" types are deduced by the compiler based on your code

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




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

- A lambda expression is a *closure* that stores a function with a data environment
- It is like a functor, but much easier to use

C++ lambda expressions…

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

- A lambda expression has the following form

```
[capture list] (parameter list) {function body}
```

Lambda expression concepts…

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

- A lambda expression has the following form

```
[capture list] (parameter list) {function body}
```

- The capture list specifies how variables (in enclosing scope) are pulled into the lambda data environment
  - Value or reference (**[=]** vs. **[&]**)? By-value is required for GPU execution, RAJA reductions, etc.
  - **We recommend using capture by-value in all cases**, as shown above

Lambda expression concepts…

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

- A lambda expression has the following form

```
[capture list] (parameter list) {function body}
```

- The capture list specifies how variables are pulled into the lambda data environment
  - We recommend using capture by-value in all cases
- The parameter list arguments are passed to lambda function body – (`int i`)

Lambda expression concepts…

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

- A lambda expression has the following form

```
[capture list] (parameter list) {function body}
```

- The capture list specifies how variables are pulled into the lambda data environment
  - We recommend using capture by-value in all cases
- The parameter list arguments are passed to lambda function body; e.g., (**`int i`**)
- A lambda passed to a CUDA kernel requires a *device annotation*:

[=] **__device__** (…) { … }

Lambda expression concepts…

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

- A lambda expression has the following form

```
[capture list] (parameter list) {function body}
```

- The capture list specifies how variables (outer scope) are pulled into lambda data environment
  - We recommend using capture by-value in all cases
- The parameter list are arguments passed to lambda function body; e.g., `(int i)` is "loop variable"
- A lambda passed to a CUDA kernel requires a device annotation: [=] **__device__** (…) { … }

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

“Bring your own” memory management

- RAJA does not provide a memory model. This is by design.
  - Users must handle memory space allocations and transfers

“Bring your own” memory management

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

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

Are ‘a’ and ‘b’ accessible on GPU?

“Bring your own” memory management

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

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

Are ‘a’ and ‘b’ accessible on GPU?

- Some possibilities:
  - **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 RAJA.

“Bring your own” memory management

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

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

Are ‘a’ and ‘b’ accessible on GPU?

For simplicity, all RAJA exercises and examples in the repository use unified memory when compiled with GPU back-end enabled.

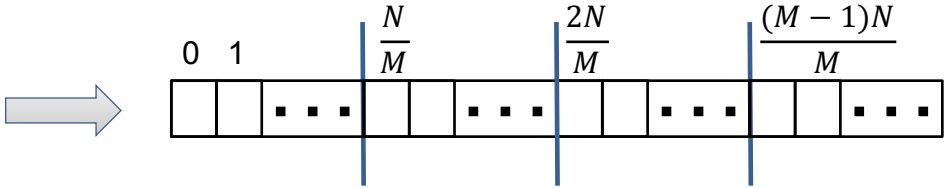
Before we get back into RAJA, we will discuss some things that are good to keep in mind…

Good to know

Parallelism comes in various forms

- Instruction-level Parallelism (ILP) – multiple machine instructions execute at the same time
- Task (functional) parallelism – tasks run concurrently on different processors based on their dependencies
- **Data Parallelism** – same operation is applied concurrently to subsets of data

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



The focus of RAJA is fine-grained, loop-level data parallelism.

Data dependencies are a key inhibitor of parallelism

Good to know

Data dependencies are a key inhibitor of parallelism

Good to know

- A data dependence occurs when multiple threads or tasks could **write to the same memory location at the same time – "race condition"**
  - This can cause an algorithm to produce non-deterministic results (order-dependent)
  - Example: a for-loop where not all loop iterations are independent

Data dependencies are a key inhibitor of parallelism

Good to know

- A data dependence occurs when multiple threads or tasks could write to the same memory location at the same time – “race condition”
  - This can cause an algorithm to produce non-deterministic results (order-dependent)
  - Example: a for-loop where not all loop iterations are independent

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

In a data parallel loop, all loop iterations are independent.

Data dependencies are a main inhibitor of parallelism

Good to know

- A data dependence occurs when multiple threads or tasks could write to the same memory location at the same time – “race condition”

```
for (int i = 0; i < N; ++i) {
   x[i] = x[i-1] + y[i];
}
```

Issue: `x[i-1]` must be computed before `x[i]` (loop-carried dependence)

Sometimes algorithms must be rewritten to enable parallelism.

Data dependencies are a main inhibitor of parallelism

Good to know

- A data dependence occurs when multiple threads or tasks could write to the same memory location at the same time – “race condition”

```
for (int r = 0; r < N; ++r) {
   for (int c = 0; c < N; ++c) {
      A[r][c] = A[c][r];
   }
}
```

Issues: A(c, r) and A(r, c) depend on each other

Tiling algorithms, using CPU stack/GPU shared memory helps enable parallelism.

Data dependencies are a main inhibitor of parallelism

Good to know

- A data dependence occurs when multiple threads or tasks could write to the same memory location at the same time – “race condition”

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

Issue: each loop iteration writes to ‘sum’

We’ll discuss RAJA reductions in this tutorial.

It’s important to measure parallel performance and know what to expect

Amdahl's law tells us the theoretical maximum "speedup" we can achieve in a parallel program

Good to know

- Amdahl's law:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

$S(n)$ is the theoretical maximum speedup of a workload run in parallel with *n* processors

p is the proportion of sequential run time (1 processor) of the parts of the program that can run in parallel

Amdahl's law tells us the theoretical maximum "speedup" we can achieve in a parallel program

Good to know

- Amdahl's law:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

$S(n)$ is the theoretical maximum speedup of a workload run in parallel with *n* processors

p is the proportion of sequential run time (1 processor) of the parts of the program that can run in parallel

- Theoretical speedup may increase as we use more processors. But, for a fixed workload, we cannot continue to add processors and expect additional speedup.

Amdahl's law tells us the theoretical maximum "speedup" we can achieve in a parallel program

Good to know

- Amdahl's law:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

$S(n)$ is the theoretical maximum speedup of a workload run in parallel with *n* processors

p is the proportion of sequential run time (1 processor) of the parts of the program that can run in parallel

- Theoretical speedup may increase as we use more processors. But, for a fixed workload, we cannot continue to add processors and expect additional speedup.

For example, if only 50% of an application can run in parallel (p = 0.5), Amdahl's law tells us the maximum speedup we could observe on any number of processors is **2X.**

Takeaway: speedup is always limited by sequential portions of your code

Good to know

- Amdahl's law:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

$S(n)$ is the theoretical maximum speedup of a workload run in parallel with *n* processors

p is the proportion of sequential run time (1 processor) of the parts of the program that can run in parallel

- Note the following:

$$S(n) \leq \frac{1}{(1-p)} \qquad \lim_{n\to\infty} S(n) = \frac{1}{(1-p)}$$

"p" is the limiting factor.

How do you know what you actually gain from parallelism?

Good to know

- Measure and compare sequential run time and parallel run time
  - "Parallel Speedup" $S_n = T_1 / T_n$
  - "Parallel Efficiency" $E_n = S_n / n$

T_1 is sequential run time

T_n is run time using n processes or threads

$S_n = n$ ($E_n = 1$) ⇒ "Perfect (ideal) scaling"

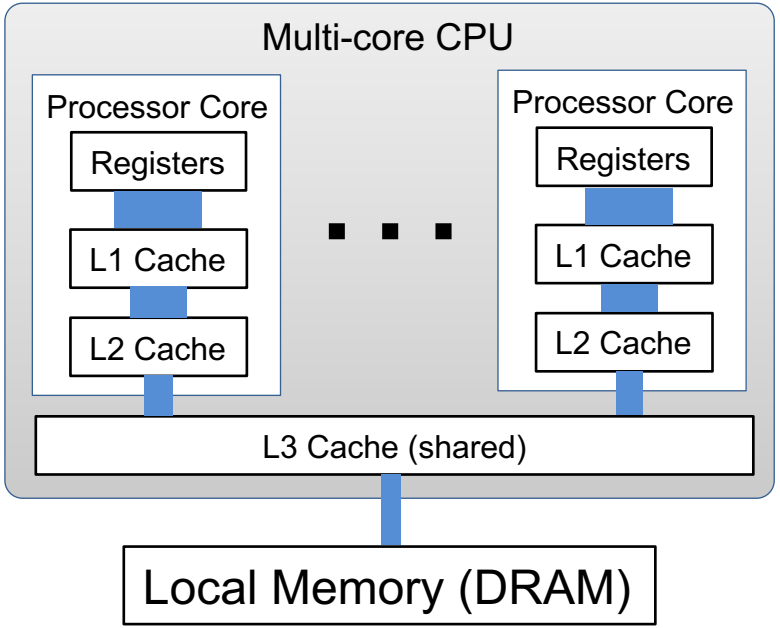
Most of the time, you will see $S_n < n$ ($E_n < 1$). However, it is possible to see $S_n > n$.

Understanding memory hardware configuration helps to program for good performance

Modern multi-core CPUs have a hierarchy of memory levels

Good to know

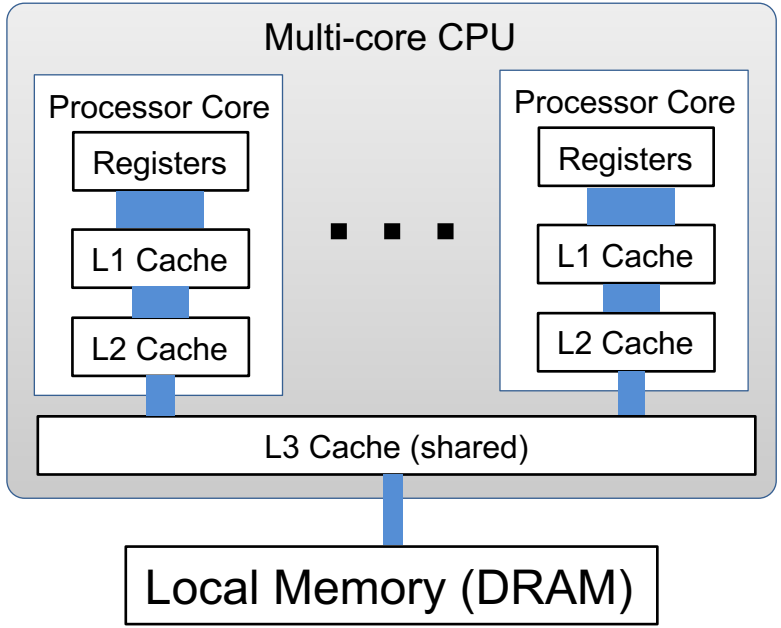
- Some are local to each core: registers, caches
- Some memory is shared with other cores or CPUs: caches, node local memory



Modern multi-core CPUs have a hierarchy of memory levels

Good to know

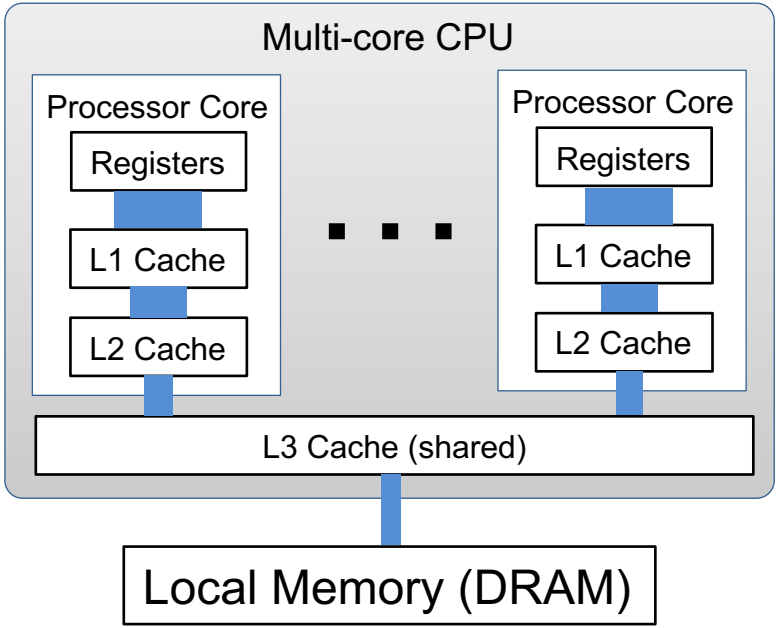
- Some are local to each core: registers, caches
- Some memory is shared with other cores or CPUs: caches, node local memory
- Data move through the memory hierarchy to each processor core as they are used and migrate away when not used



Modern multi-core CPUs have a hierarchy of memory levels

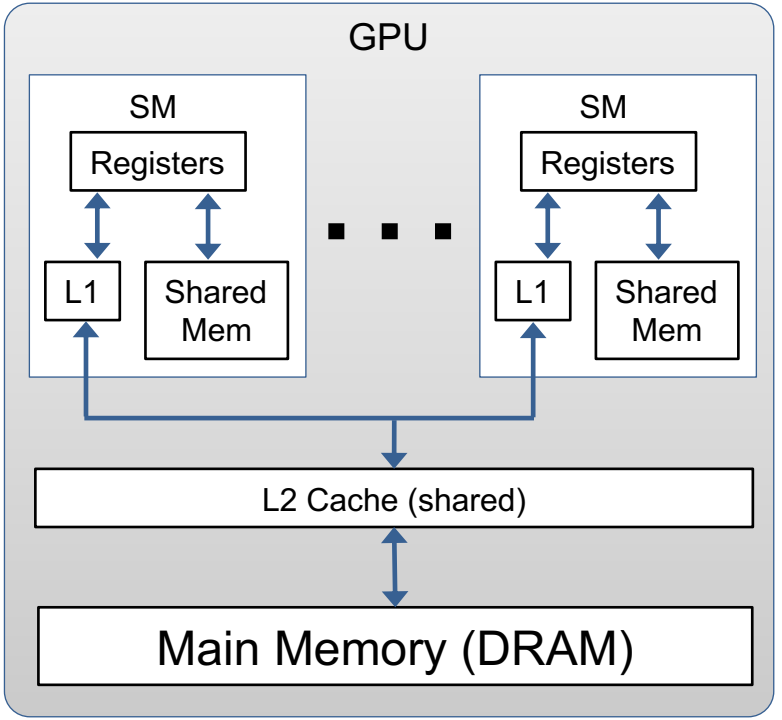
Good to know

- Some are local to each core: registers, caches
- Some memory is shared with other cores or CPUs: caches, node local memory
- Data move through the memory hierarchy to each processor core as they are used and migrate away when not used
- Memory capacity and access times increase significantly as you get farther away from the processors
- Levels closer to a core have higher **bandwidth** (speed) and lower **latency** (delay)



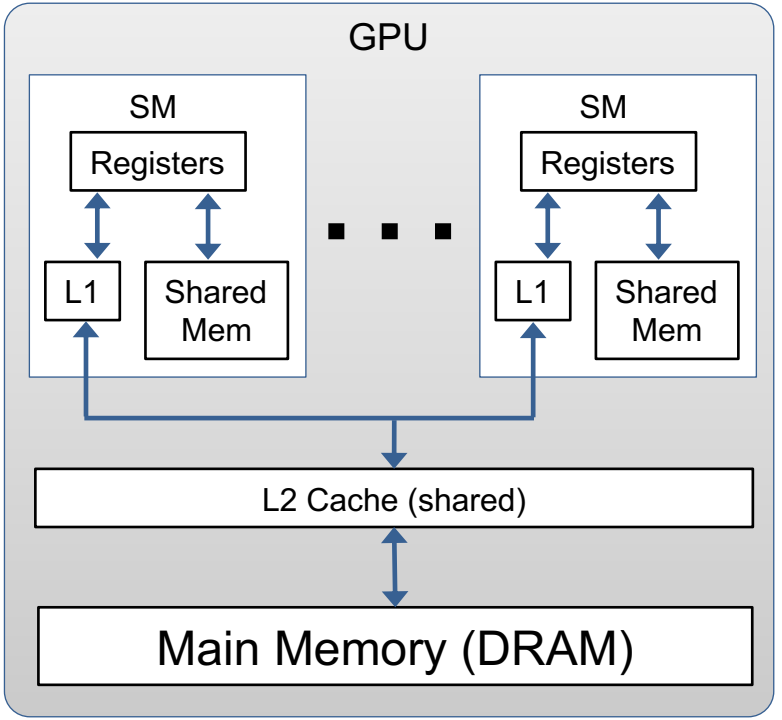
GPUs also have a memory hierarchy

- Some memory levels are local to each streaming multiprocessor (SM), some are shared by SMs



GPUs also have a memory hierarchy

- Some memory levels are local to each streaming multiprocessor (SM), some are shared by SMs
- Each SM has a register file, memory for L1 cache or shared memory (accessible by all threads in each thread block)
  - These have high bandwidth and very low latency
- A unified cache (L2) is shared by all SMs
- Main memory (DRAM) is accessible by GPU and host CPU (e.g., host-device copy)



Reducing memory motion is critical for good performance

Good to know

- General “rules of thumb”
  - Place data that are **used together close in memory**: think locality – spatial and temporal
  - Consider **data access patterns** when designing algorithms

Reducing memory motion is critical for good performance

Good to know

- General “rules of thumb”
  - Place data that are used together close in memory: think locality – spatial and temporal
  - Consider data access patterns when designing algorithms
- **Memory coalescing** is very important for GPU performance
  - Multiple memory accesses are combined into one memory transaction
  - With CUDA, you typically want all 32 threads in a warp to read operands & write results in as few transactions as possible and avoid serialized memory access
  - **Avoid memory accesses that are non-sequential, sparse, or misaligned**
- Useful references:
  - “What Every Programmer Should Know About Memory” by Ulrich Drepper (https://akkadia.org/drepper/cpumemory.pdf)
  - “Introduction to GPGPU and CUDA Programming” by Philip Nee (https://cvw.cac.cornell.edu/GPU/default)

Back to RAJA…

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 hide 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 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 reduced value inside a kernel. This may change in the future.

- 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::tbb_reduce;
RAJA::omp_target_reduce;
```

Sample RAJA reduction policy types.

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

RAJA supports five common reductions types

```
RAJA::ReduceSum<    REDUCE_POLICY, DTYPE > r(in_val);

RAJA::ReduceMin<    REDUCE_POLICY, DTYPE > r(in_val);

RAJA::ReduceMax<    REDUCE_POLICY, DTYPE > r(in_val);

RAJA::ReduceMinLoc< REDUCE_POLICY, DTYPE > r(in_val,
                                             in_loc);

RAJA::ReduceMaxLoc< REDUCE_POLICY, DTYPE > r(in_val,
                                             in_loc);
```

Initial "loc" values

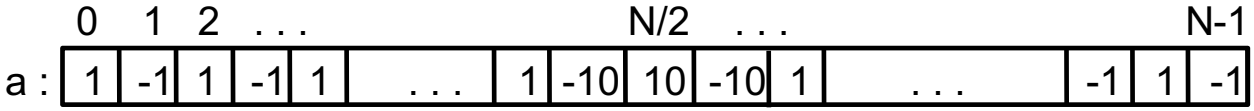
"Loc" reductions give a loop index where reduced value was found.

Multiple RAJA reductions can be used in a kernel

```
RAJA::ReduceSum< REDUCE_POL, int > sum(0);
RAJA::ReduceMin< REDUCE_POL, int > min(MAX_VAL);
RAJA::ReduceMax< REDUCE_POL, int > max(MIN_VAL);
RAJA::ReduceMinLoc< REDUCE_POL, int > minloc(MAX_VAL, -1);
RAJA::ReduceMaxLoc< REDUCE_POL, int > maxloc(MIN_VAL, -1);

RAJA::forall< EXEC_POL >( RAJA::RangeSegment(0, N), [=](int i) {
  seq_sum += a[i];

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

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

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

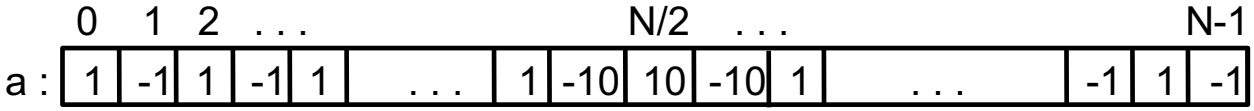
‘a’ is an int vector of length ‘N’ (N / 2 is even) initialized as:



- *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:



- *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)

Generally, the result of a parallel reduction is order-dependent.

Let’s work through some exercises

If you want to work exercises on your own…

- Files are located in the RAJA/exercises/tutorial_halfday directory **(use v0.11.0 release)**
  - Each exercise has two files – one to work through (fill in code sections) and one with a solution
- To work through an exercise….
  - The associate 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 the 'build' directory)
  - Run the exercise executable file (i.e., located in 'build/bin' directory)
  - Check the output to see if what you did passes or fails the checks

In the interest of time, we will talk through the exercises as a group today.
Note that the online RAJA User Guide provides useful information to help you work the exercises.

Exercise #1: vector addition

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

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

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

- *Exercise*: *Implement RAJA sequential, OpenMP, and CUDA variants of vector addition. Code sections for you to fill in are indicated in comments in the file noted above. Run the code and check your results. The 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.

Exercise #1 solution

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** contains a complete implementation of 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 >(RAJA::RangeSegment(0, N),
  [=] (int i) {
    c[i] = 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 *Riemann integration*.

- *Exercise: Implement RAJA variants for sequential, OpenMP, and CUDA using RAJA reductions. Code sections for you to fill in are indicated in comments in the file noted above. The file also 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 #2 solution

- File **RAJA/exercises/tutorial_halfday/ex2_approx-pi_solution.cpp** contains solution to the exercise. For example, OpenMP versions:

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 the policies…

```
RAJA::seq_exec – RAJA::seq_reduce

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

Iteration spaces : Segments and IndexSets

A RAJA "Segment" defines a loop iteration space

Iteration spaces

- 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)

Loop iteration spaces are defined by Segments

Iteration spaces

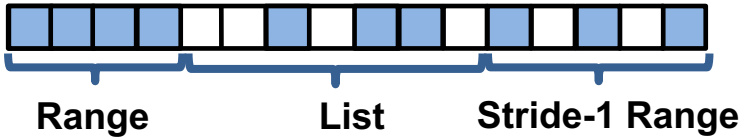
- A Segment defines a set of loop indices to run in a kernel

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)

Iteration spaces

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

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

Runs loop indices: 0, 1, 2, … , N-1

A RangeStrideSegment defines a strided sequence of indices


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

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

Runs loop indices: 0, 2, 4, …

RangeStrideSegments also support negative indices and strides


```
RAJA::RangeStrideSegment srange2( N-1, -1, -1 );
```

```
RAJA::forall< RAJA::seq_exec >( srange2 , [=] (int i)
{
   // ...
} );
```

Runs loop in reverse: N-1, N-2, … , 1, 0

Segments are templates on the index type

RangeSegment and RangeStrideSegment are **type aliases**

```
using RAJA::RangeSegment =
        RAJA::TypedRangeSegment<RAJA::Index_type>;

using RAJA::RangeStrideSegment =
        RAJA::TypedRangeStrideSegment<RAJA::Index_type>;
```

Segments are templates on the index type

RangeSegment and RangeStrideSegment are **type aliases**

```
using RAJA::RangeSegment =
        RAJA::TypedRangeSegment<RAJA::Index_type>;

using RAJA::RangeStrideSegment =
        RAJA::TypedRangeStrideSegment<RAJA::Index_type>;
```

- RAJA::IndexType is a useful parametrization
  - It is an alias to std::ptrdiff_t
  - **Appropriate for most compiler optimizations**

Use the 'Typed' Segment types for other index value types.

A ListSegment can define any set of indices

Iteration spaces

```
using IdxType = RAJA::Index_type;
using ListSegType = RAJA::TypedListSegment<IdxType>;

// array of indices
IdxType idx[ ] = {10, 11, 14, 20, 22};

// ListSegment object containing indices...
ListSegType idx_list( idx, 5 );
```

Think *“indirection array”.*

A ListSegment can define any set of indices

Iteration spaces

```
using IdxType = RAJA::Index_type;
using ListSegType = RAJA::TypedListSegment<IdxType>;

// array of indices
IdxType idx[ ] = {10, 11, 14, 20, 22};

// ListSegment object containing indices...
ListSegType idx_list( idx, 5 );

RAJA::forall< RAJA::seq_exec >( idx_list, [=] (IdxType i)
{
  a[i] = ...;
} );
```

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

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

A RAJA IndexSet is a container of Segments

Iteration spaces

```
using RangeSegType = RAJA::TypedRangeSegment<RAJA::Index_type>;
using ListSegType  = RAJA::TypedListSegment<RAJA::Index_type>;

RangeSegType range1(0, 8);

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

RangeSegType range3(24, 28);
```

An IndexSet may contain different Segment types

Iteration spaces

```
using RangeSegType = RAJA::TypedRangeSegment<RAJA::Index_type>;
using ListSegType = RAJA::TypedListSegment<RAJA::Index_type>;

RangeSegType range1(0, 8);

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

RangeSegType range3(24, 28);

RAJA::TypedIndexSet< RangeSegType, ListSegType > iset;

iset.push_back( range1 );
iset.push_back( list2 );
iset.push_back( range3 );
```

Segment types must be specified at compile time

IndexSets enable iteration space partitioning

Iteration spaces

```
using RangeSegType = RAJA::TypedRangeSegment<RAJA::Index_type>;
using ListSegType = RAJA::TypedListSegment<RAJA::Index_type>;

RangeSegType range1(0, 8);

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

RangeSegType range3(24, 28);

RAJA::TypedIndexSet< RangeSegType, ListSegType > iset;

iset.push_back( range1 );
iset.push_back( list2 );
iset.push_back( range3 );
```

Iteration space is partitioned into 3 Segments

0, …, 7	,	10, 11, 14, 20, 22	,	24, …, 27
range1		list2		range3

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


```
using ISET_EXECPOL =
   RAJA::ExecPolicy< RAJA::omp_parallel_segit,
                     RAJA::seq_exec >;

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

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

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

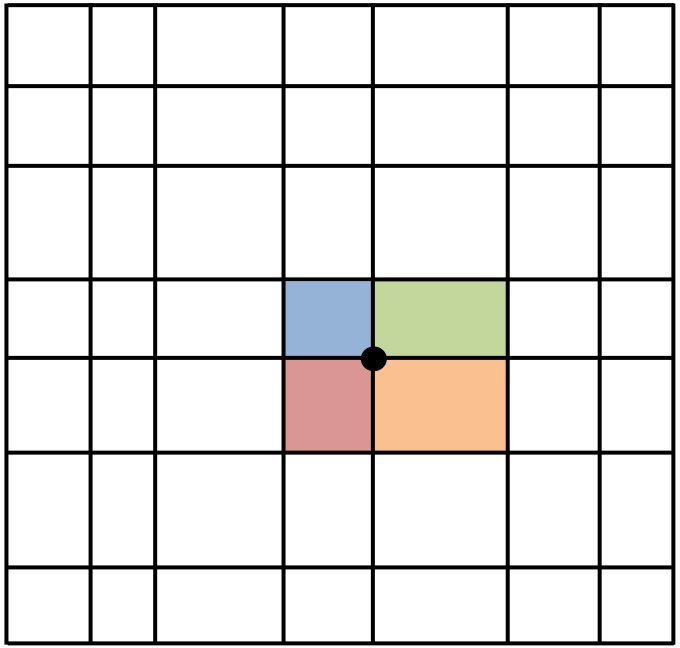
Why does RAJA provide Index Sets?

- **Multiphysics codes use indirection arrays** (a lot!)
  - Indirection inhibits performance: more instructions + memory traffic, impedes optimizations
- **Range Segments are better for performance**
  - When large stride-1 ranges are embedded in an iteration space…
  - …you can expose these as SIMD-izable ranges "in place" to compilers (no gather/scatters)
- **Partitioning and reordering iterations gives flexibility and performance**
  - Avoid fine-grained synchronization (atomics or critical sections), which are **contention heavy**
  - Avoid extra arrays and gather/scatter operations, which require **extra memory traffic**
  - Prefer coarse-grained synchronization, which has much **lighter memory contention**

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

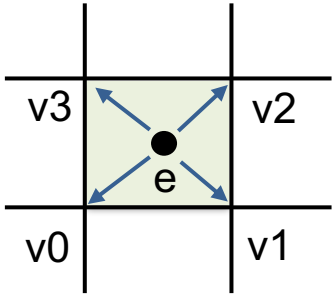
IndexSets can help enable parallelism

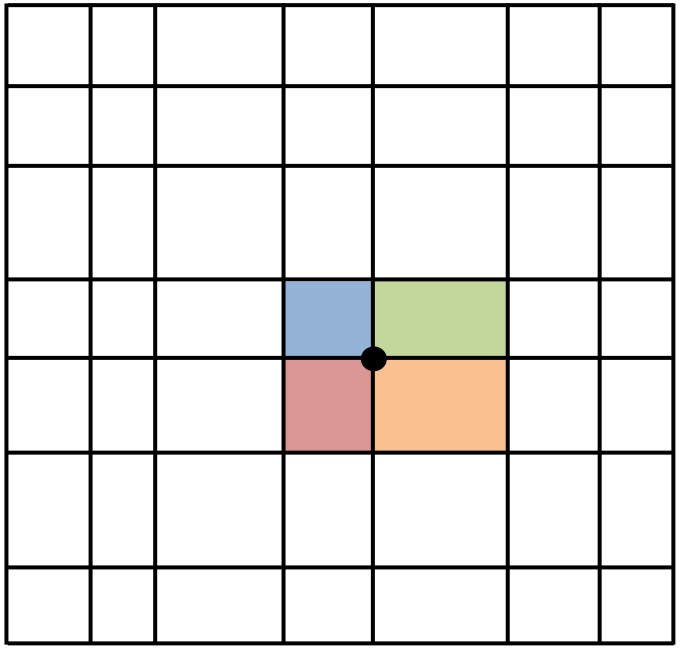
- Consider an irregularly-spaced 2D Cartesian mesh
- At each mesh vertex, we want to store the average area of the 4 surrounding elements



IndexSets can help enable parallelism

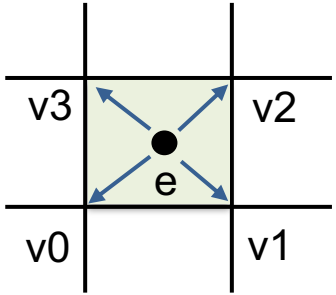
- Consider an irregularly-spaced 2D Cartesian mesh
- At each mesh vertex, we want to store the average area of the 4 surrounding elements
  - For each element e, add ¼ area(e) to area(vi), i = 0,…,3





A C-style serial code for the vertex area

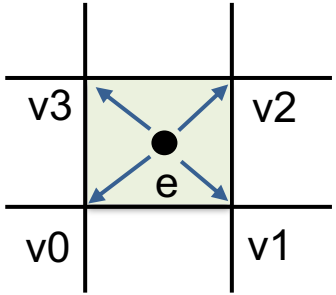
```
for (int ie = 0 ; ie < N_elem ; ++ie) {

    int* iv = e2v_map(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 ;

}
```



As written, will this code be correct when run in parallel?

A C-style serial code for the vertex area

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

    int* iv = e2v_map(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 ;

}
```

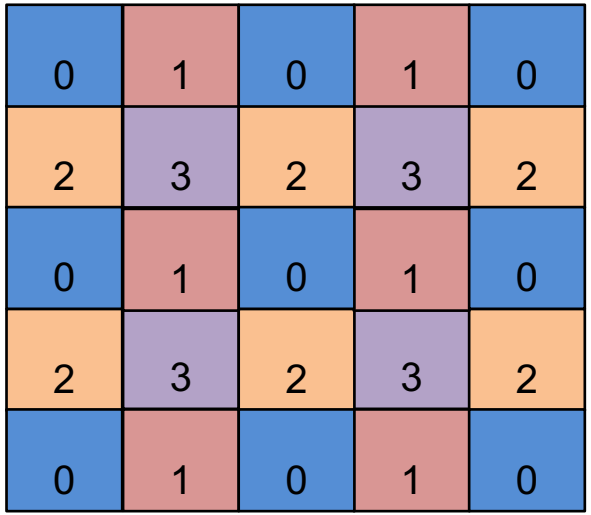


As written, will this code be correct when run in parallel?

No. There is a data race at each vertex.

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

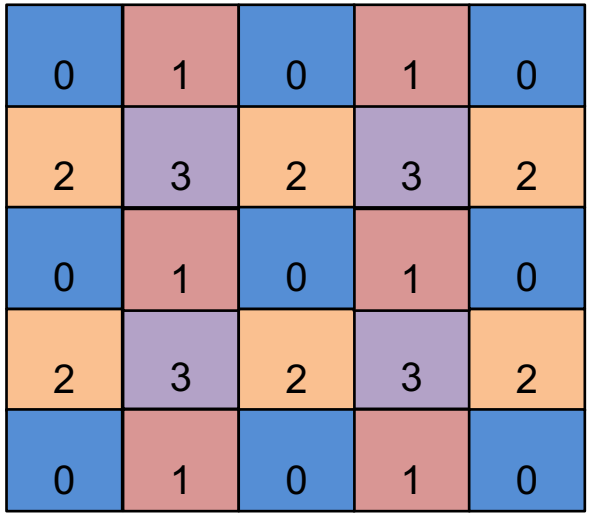

```
for (int ie = 0 ; ie < N_elem ; ++ie) {

    int* iv = e2v_map(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 the elements into four subsets (colors) and run each in parallel


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

    int* iv = e2v_map(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 ;

}
```

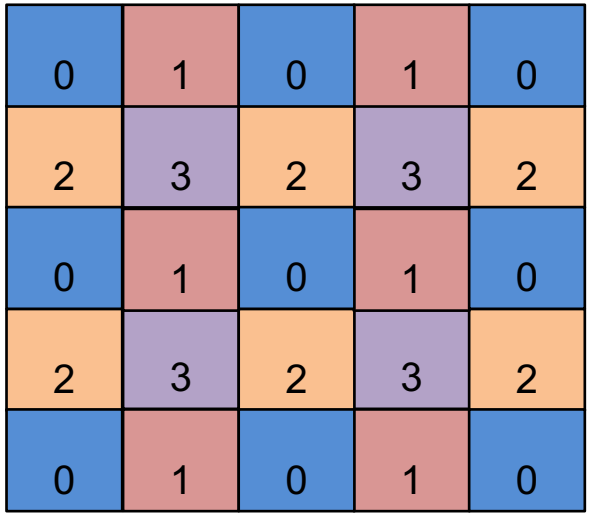


Will the results be reproducible?

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

Exercises

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

    int* iv = e2v_map(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 the results be reproducible?

Yes. The computation for all elements with same color (number) is data parallel.

Exercise #3: Mesh vertex area using "colored" index set

- File **RAJA/exercises /tutorial_halfday/ex3_colored-indexset.cpp** contains a C-style sequential implementation of the vertex area calculation. It also contains a C-style OpenMP variant that uses arrays to enumerate the elements of each color and partition the operation into independent subsets:

```
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: Implement RAJA OpenMP and CUDA variants of the vertex area calculation using a RAJA IndexSet with 4 ListSegments. The file noted above contains RAJA IndexSet execution policy types for each case and empty code sections for you to fill in. It also has methods you can use to check your work and print results.

See the RAJA User Guide for a description of RAJA index set execution policies.

Exercise #3 solution

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

- The RAJA code for defining the 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 ListSegments that define the element indices for each of the four colors (also used in the non-RAJA variant shown on the previous 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
```

Atomic operations

RAJA provides portable atomic operations

```
// two pointers to 'int' memory locations
int* x = ...
int* y = ...

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

  RAJA::atomicAdd< ATOMIC_POLICY >(x, 1); // atomically add 1 to x
  RAJA::atomicSub< ATOMIC_POLICY >(y, 1); // atomically subtract
                                          // 1 from y

} );
```

An atomic operation updates a specific memory address (write or read-modify-write) where only one thread or process at a time can write to it.

Recall exercise #2

- We approximated π using Riemann integration and the following 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 could also use an atomic operation to prevent multiple threads from attempting to write to the memory address of 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;
```

The atomic policy must be compatible with the loop execution policy (similar to reductions).

The RAJA "builtin" atomic policy uses compiler built-in atomics

```
using EXEC_POL = RAJA::omp_parallel_for_exec;

int *sum = ...;

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

  RAJA::atomicAdd< RAJA::builtin_atomic >(sum, 1);

} );
```

The RAJA “auto” atomic policy will pick the correct atomic implementation

```
using EXEC_POL = RAJA::omp_parallel_for_exec;

int *sum = ...;

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

  RAJA::atomicAdd< RAJA::auto_atomic >(sum, 1);

} );
```

Some may prefer this option for simpler portability.

RAJA also has an interface modeled after the C++20 standard std::atomic_ref feature

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

For example:

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

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

Result: sum is 5 (= 2 + 1 + 1 + 1).

RAJA provides a variety of atomic operations

- Arithmetic: add, sub
- Min, max
- Increment/decrement: inc, dec, including conditional comparisons with other values
- Bitwise-logical: and, or, xor
- Replace: exchange, compare-and-swap (CAS)
- C++ std::atomic_ref style interface (RAJA::AtomicRef)

The RAJA User Guide describes the full set of RAJA atomics.

Exercise #4: atomic histogram

- The file **RAJA/exercises/tutorial_halfday/ex4_atomic-histogram.cpp** contains C-style sequential and OpenMP implementations of a histogram calculation:
  - Given an integer array of length N with entries in the set {0, 1, 2, …, M-1}, where M < N. Build 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

- *Exercise: Implement RAJA sequential, OpenMP, and CUDA loops to compute the histogram array using RAJA atomic operations. The file noted above contains empty code sections for you to fill in and methods you can use to check your work and print results.*

See the RAJA User Guide for a listing of RAJA loop execution and atomic policies.

Exercise #4 solution

- The file **RAJA/exercises/tutorial_halfday/ex4_atomic-histogram_solution.cpp** contains a complete implementation of exercise #4.

- OpenMP:

C-style

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

The RAJA sequential and CUDA versions are similar but use **seq_atomic** and **cuda_atomic** policies. Alternatively, the RAJA **auto_atomic** policy could be used.

RAJA Version

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

Scan operations

Scan is an important building block for parallel algorithms

- It is a key primitive for developing parallel algorithms
  - Based on reduction tree and reverse reduction tree
  - Scan is an example of a computation that looks inherently serial, but for which there exist efficient parallel implementations

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

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

Prefix sum is the most common scan operation

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

RAJA::inclusive_scan< EXEC_POL >(in, in + N, out);

RAJA::exclusive_scan< EXEC_POL >(in, in + N, out);
```

Example:

In : 8 -1 2 9 10 3 4 1 6 7 (N=10)

Out (inclusive) : 8 7 9 18 28 31 35 36 42 49

Out (exclusive) : 0 8 7 9 18 28 31 35 36 42

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 “+”.

The output array contains partial sums of input array.

RAJA also provides “in-place” scan operations

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

```
RAJA::inclusive_scan_inplace< EXEC_POL >(arr, arr + N);
```

```
RAJA::exclusive_scan_inplace< EXEC_POL >(arr, arr + N);
```

“In-place” scans return the result in the input array.

RAJA provides different operators to use in scans

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

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

Out : 2147483648 8 -1 -1 -1 -1 -3 -3 -3 -3

What is the first value in the result of this scan?

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

RAJA provides different operators to use in scans

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

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

Out : 2147483648 8 -1 -1 -1 -1 -3 -3 -3 -3

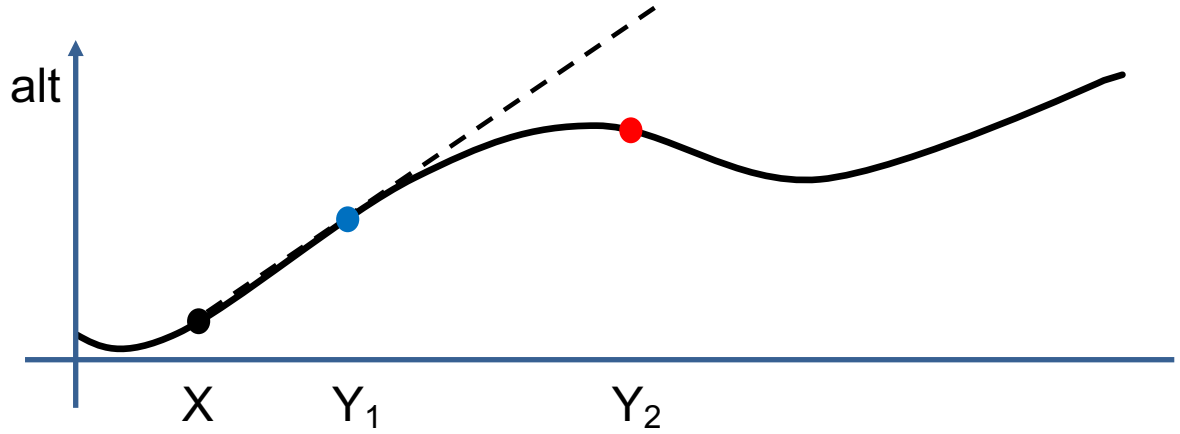
What is the first value in the result of this scan?

It is the identity of the minimum operator – min(I, value) = value

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

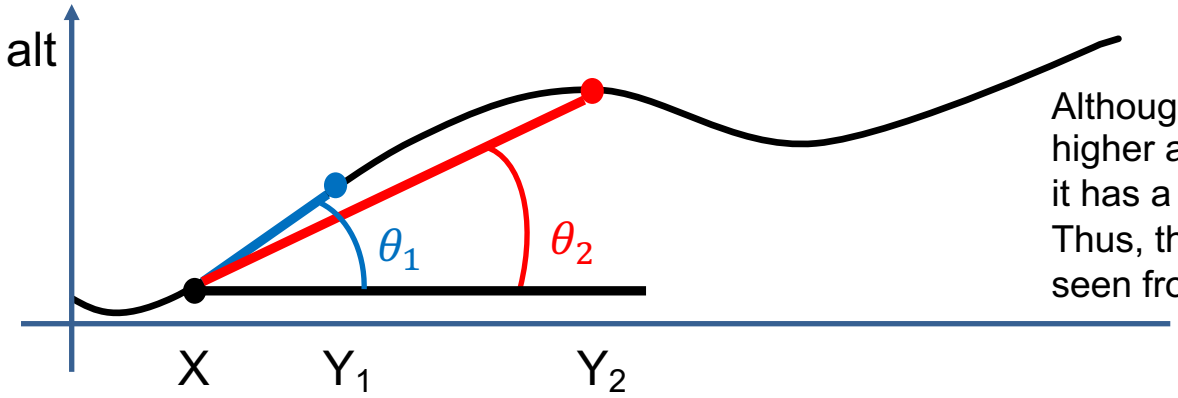
Exercise #5: the line-of-sight problem

- 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

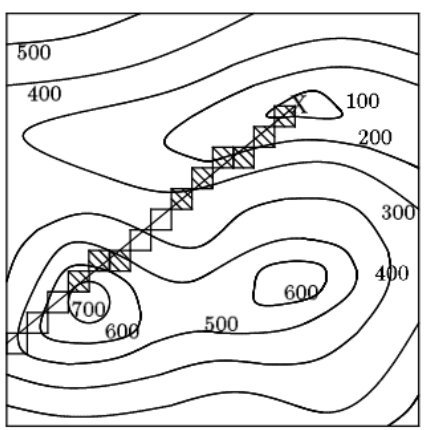
- The *line-of-sight* problem: given an observation point at X on a terrain map, and a set of points along a ray starting at X, find which points on the terrain are visible from X.
- A point at Y on the surface is visible from the point at X if and only if no other point on the terrain between the points X and Y has a greater vertical angle from X than Y.



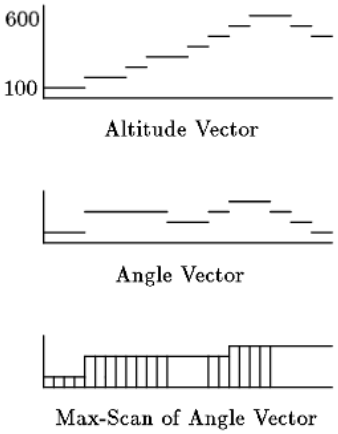
Although the point at Y_2 has a higher altitude than the point at Y_1, it has a smaller vertical angle. Thus, the point at Y_2 cannot be seen from the point at X.

Exercise #5: the line-of-sight problem

- A point at Y on the ray 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.
- Let ‘altX’ be the altitude at point X and let ‘alt’ be a vector defined so that alt[i] is the altitude at point Y_i.
- Let ‘dist’ be a vector defined so that dist[i] is the horizontal distance between point X and point Y_i.
- We compute an angle vector ‘ang’ such that ang[i] is the vertical angle at the 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’ that we can use 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 code that implements the line-of-sight algorithm described on the previous slide. It looks like this:

```
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 RAJA scan operations to compute the max angle scan vector and RAJA::forall loops to determine which points are visible. The file noted above contains empty code sections for you to fill in and methods you can use to check your work and print results.*

See the RAJA User Guide for a listing of RAJA scan execution policies and operators.

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 RAJA scan operations use the same execution policies as RAJA::forall.

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 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

View/Layout

- 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

View/Layout

- 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 data indexing 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);
}
```

RAJA Views and Layouts support any number of dimensions

View/Layout

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

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

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

// iterate over jth index and hold others fixed
for (int j = 0; j < Nj; ++j) {
  Aview(i0, i1, ..., j, ..., iN) = ...;
}
```

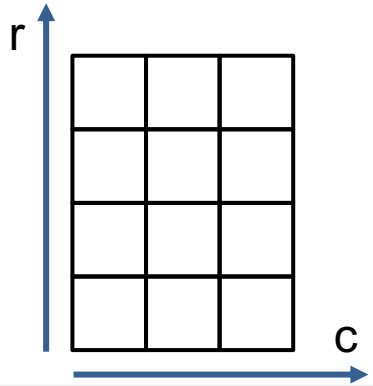
Stride-1 data access

Data access stride is Nn * … * N(j+1)

The RAJA default layout uses ‘row-major’ ordering.
So, the right-most index is stride-1 when using the Layout<DIM>.

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)

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[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 is subtracted from each loop index to access data.

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

Important notes about RAJA Layout types

- Since each Layout object has a permutation, there is no ``RAJA::PermutedLayout`` type:

```
RAJA::Layout< NDIMS > perm_layout =
  RAJA::make_permuted_layout( ... );
```

- An offset layout has a ``RAJA::Layout`` and offset data. So ``RAJA::OffsetLayout`` is a distinct type:

```
RAJA::OffsetLayout< NDIMS > offset_layout =
  RAJA::make_offset_layout( ... );
```

```
RAJA::OffsetLayout< NDIMS > perm_offset_layout =
  RAJA::make_permuted_offset_layout( ... );
```

Offset layout quiz…

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

- *What index space does this layout represent?*

Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

Offset layout quiz…

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

- *What index space does this layout represent?*

  The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Index ‘1’ (right-most) is stride-1 (using default permutation).

Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Index ‘1’ (right-most) is stride-1 (using default permutation).

- *What is the stride of index ‘0’?*

Offset layout quiz…

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

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5].

- *Which index is stride-1?*

Index ‘1’ **(right-most)** is stride-1 (default permutation).

- *What is the stride of index ‘0’?*

Index ‘0’ has stride 11 (since index 1 has extent 11, [-5, 5]).

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, -5}}, {{2, 5}}, 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, -5}}, {{2, 5}}, perm );
```

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

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, -5}}, {{2, 5}}, perm );
```

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *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, -5}}, {{2, 5}}, perm );
```

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

Index '0' 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, -5}}, {{2, 5}}, perm );
```

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to permutation).

- *What is the stride of index '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, -5}}, {{2, 5}}, perm );
```

- *What index space does this layout represent?*

The 2-d index space [-1, 2] X [-5, 5] (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to permutation).

- *What is the stride of index '1'?*

Index '1' has stride 4 (since index '0' has extent 4, [-1, 2]).

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

View/Layout

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

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

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

View/Layout

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

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

```
// Convert i=2, j=3, k=1 to linear index
int lin = layout(2, 3, 1);
```

What is the value of "lin"?

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

View/Layout

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

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

```
// Convert i=2, j=3, k=1 to linear index
int lin = layout(2, 3, 1);
```

What is the value of "lin"?

lin = 188 (= 1 + 3 * 11 + 2 * 11 * 7)

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


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

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

```
// Convert linear index 191 to 3d (i,j,k) index
layout.toIndices(191, i, j, k);
```

What is the 3d index tuple (i, j, k)?

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


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

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

```
// Convert linear index 191 to 3d (i,j,k) index
layout.toIndices(191, i, j, k);
```

What is the 3d index tuple (i, j, k)?

(i, j, k) = (2, 3, 4)

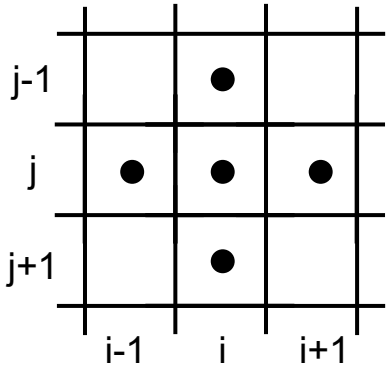
191 (= **4** + **3** * 11 + **2** * 11 * 7)

RAJA provides a compile configuration option to check at run time whether indices are in bounds. See the User Guide for details.

Exercise #6: 5-point stencil

- Consider a simple “five-point stencil” computation 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 for each element on the mesh interior:

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

and the “B” array has an entry 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

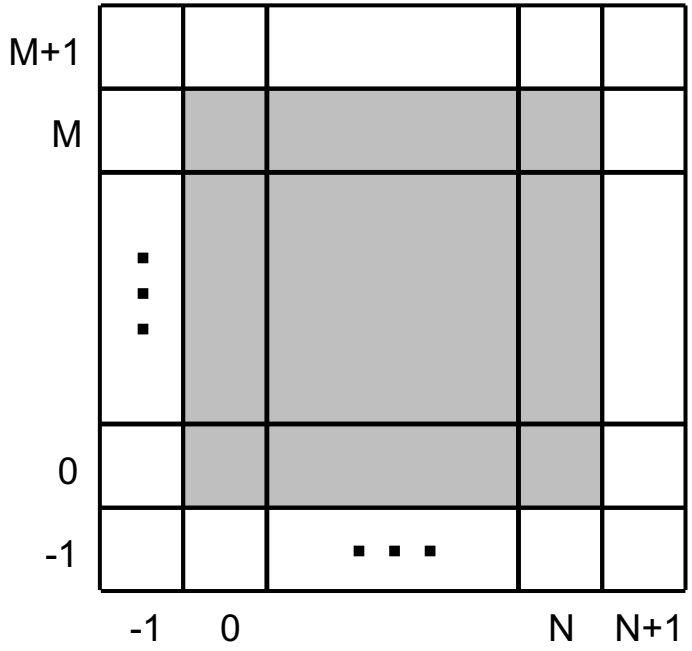
- That is, B has a value for each element on the mesh to the right and A has a value for each element in the grey interior region.

- 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 formula:

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



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.
  - Part A assumes that the column index (j-loop) is stride-1.
  - Part B assumes that the row index (i-loop) is stride-1.
  - Note that the manual index offset arithmetic is different for these

- *Exercise: Implement sequential variants of parts A and B using RAJA Views. The file noted above contains empty code sections to fill. The goal of this exercise is for you to learn the mechanics of creating and using RAJA Layouts and Views so RAJA execution methods are not essential and are not used. The file contains methods you can use to check your work and print results.*

- Note: with RAJA Views, you can make the loop bodies look 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)
```

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.
  - Part A: column index (j-loop) is stride-1. Part B: row index (i-loop) is stride-1.
- Here's the C-style version for Part B:

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

    int idx_out = i + Nc_int * j;
    int idx_in = (i + 1) + Nc_tot * (j + 1);

    A_ref[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

  }
}
```

The implementation for Part A differs only in the index offset arithmetic.

Exercise #6: 5-point stencil

- *Exercise: Implement sequential variants of the stencil computation (parts A and B):*
  - *First, define appropriate RAJA::Views for the arrays using offsets and permutations as needed for each part*
  - *Second, use the Views in a C-style loop nest to perform the computation for each part*
- 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);
```

The A part is similar, but without the permutation.

Exercise #6 solution

- The loops to do 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 there is no permutation in the Views 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 file).

Complex Loops and Advanced RAJA Features

Nested Loops

Let's look at 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[k + N*row] * B[col + N*k];
    }
    C[col + N*row] = dot;

  }
}
```

Consider using nested ‘forall’ statements for a RAJA implementation…

Nested loops

```
RAJA::forall< exec_policy_row >( row_range, [=](int row) {

  RAJA::forall< exec_policy_col >( col_range, [=](int col) {

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

  } );
} );
```

Note: we use RAJA Views to simplify multi-dimensional indexing.

…This doesn't work well

- ***Each loop level is treated as an independent entity***
  - So parallelizing the row and column loops together is hard
- We can parallelize the outer row loop (OpenMP, CUDA, etc.)
  - But then, each thread executes all code in the inner two loops sequentially
- Parallelizing the inner column loop introduces unwanted synchronization
  - We launch a new parallel computation for each row
- Loop interchange and other transformations require changing the source code of the kernel (which breaks RAJA encapsulation)

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

The RAJA::kernel API is designed for composing and transforming complex parallel 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 before.

The RAJA::kernel interface uses four basic concepts

Nested loops

- These are 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

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 ) {

  // ...
} );
```

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 ) {
  // ...
} );
```

‘0’ → col
‘1’ → row

Integer parameter in each ‘For’ statement indicates the iteration space tuple item it applies to.

To reorder the loops, we change the execution policy, not the algorithm 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.

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

Nested loops

- 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 provides various RAJA::statement types

- We will discuss several of them in this tutorial
- See the RAJA User Guide for a complete listing of available statement types and how they work

Exercise #7: Nested loop reordering

- File **RAJA/exercises/ex7_nested-loop-reorder.cpp** contains a C-style 3-level 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 matches lambda argument list: 'i' – 0, 'j' – 1, and 'k' - 2.

The exercise uses RAJA's strongly-typed indices

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*

- The kernel should look the same for any permutation 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 file **RAJA/exercises/ex7_nested-loop-reorder_solution.cpp** contains a complete implementation of the solution to the exercise

Exercise #7 solution

- The exercise uses RAJA strongly-typed indices so that if the index arguments to the lambda expression do not match the index space tuple, the code will not compile:

```
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);
```

- As seen on the previous slide, these are used in the kernel like this:

```
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 solution

- The order of statements in the kernel policy determines the loop ordering
- 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) {
      ...
```

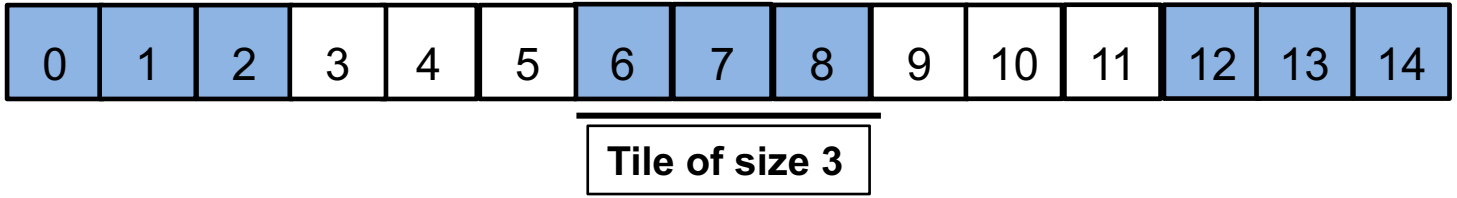
- Ordering: i outer, k middle, j inner

```
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) {
      ...
```

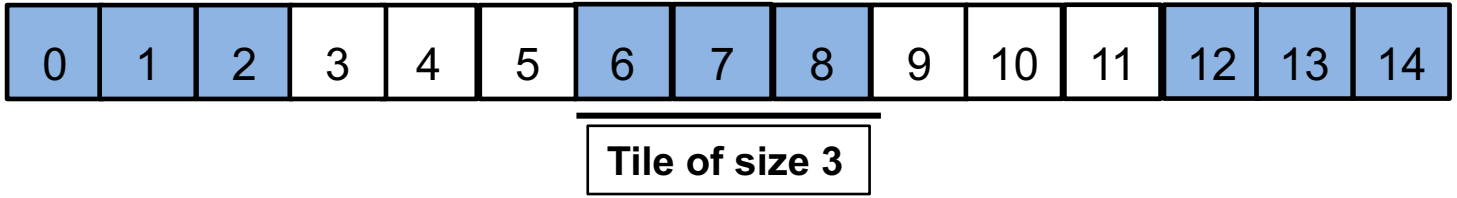
Loop Tiling

With loop tiling, data is processed in chunks



- Tiling helps ensure data used in a loop stays in a cache while it is used
- Typically different levels of memory are used to reduce costs of reads and writes

With loop tiling, data is processed in chunks




```
// standard loop
for (int id = 0; id < N; ++id) {

}
```

```
// outer loop over tiles
for (int i = 0; i < N_tile; ++i) {

  // inner loop inside a tile
  for (int ti = 0; ti < TILE_DIM; ++ti) {
    //global index
    int id = i * TILE_DIM + ti;
  }
}
```

Tile size is a performance tuning parameter.

Tiling can improve the performance of many algorithms

- Constructing a matrix transpose is an example
- Decompose a matrix into a collection of tiles, then transpose data within a tile

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \qquad \begin{pmatrix} a_{00} & a_{10} & & \\ a_{01} & a_{11} & & \\ & & & \\ & & & \end{pmatrix}$$

A A^T

Tiling can improve the performance of many algorithms

- Loop tiling improves spatial and temporal locality of data access

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

A

$$\begin{pmatrix} a_{00} & a_{10} & & \\ a_{01} & a_{11} & & \\ a_{02} & a_{12} & & \\ a_{03} & a_{13} & & \end{pmatrix}$$

A^T

Tile data may be stored in CPU stack or GPU shared memory.

C-style matrix transpose without storing local tile

$A^T(c, r) = A(r, c)$, 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;  // column index
        int row = br * TILE_SZ + tr;  // row 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 need for multiple loops, global-tile index conversion, and bounds checks

```
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

        ...
      >
    >
  >;
```

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

RAJA tiling statements eliminate need for multiple loops, global-tile index conversion, and bounds checks


```
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 non-tiled case.

Note that global indices are calculated automatically.

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:

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

    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) {
          Atview(col, row) = Aview(row, col);
        }

      }
    }
  }
}
```

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

Exercise #8: Tiled matrix transpose

- *Exercise: Implement RAJA kernel variants of the matrix transpose operation for sequential, OpenMP, and CUDA execution as described in the file. 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.*
- *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).*

Notes:
- Bounds check are not needed since RAJA tiling statements 'mask' out-of-bounds indices.
- Global indices are passed into each lambda. There is no need to compute then manually.

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 execution policy types are used in each kernel policy.

```
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>
            >
          >
        >
      >
    >;
```

Local Data

Many algorithms require non-perfectly nested loops to improve performance

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

- However, 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 unified RAJA kernel that is portable?

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<2, exec_policy1,
    For<1, exec_policy0,
      Lambda<0>
      For<0, exec_policy2,
        Lambda<1>
      >,
      Lambda<2>
    >
  >
>
```

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

RAJA::kernel_param takes an additional tuple for thread-local variables and kernel-local arrays

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 that all lambdas have the same args, but not all args must be used in each lambda.

The execution policy composes statements that define the kernel execution pattern

Local Data

```
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;


      >
    >
  >;
```

Again, nested RAJA policy statements are analogous to nested statements in a C-style loop nest.

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<1, 0>, // 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 ‘1’ and ‘0’ across CPU threads.

Policy example: launch loops as a CUDA kernel

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

Back to the matrix transpose loop tiling example…

Local Data

```
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];

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

    for (int tc = 0; tc < TILE_SZ; ++tc) {      // Write a tile of 'At'
      for (int tr = 0; tr < TILE_SZ; ++tr) {
        if (row < N_r && col < N_c) { At(col, row) = Tile[tr][tc]; }
      }
    }
// etc.
```

C-style 'tiled' loop nest

Using a local stack array improves memory access efficiency in a tile.

Different parallel strategies have different access requirements for local data

Local Data

```
Parfor (int br = 0; br < Ntile_r; ++br) {
  for (int bc = 0; bc < Ntile_c; ++bc) {

    // Thread-private array
    int Tile[TILE_DIM][TILE_DIM];

    for (int tr = 0; tr < TILE_DIM; ++tr) {
      for (int tc = 0; tc < TILE_DIM; ++tc) {
        Tile[tr][tc] = A(row, col);
      }
    }

    // ...
  }
}
```

When the outer loop is parallel, tile data should be private to each thread

```
for (int br = 0; br < Ntile_r; ++br) {
  for (int bc = 0; bc < Ntile_c; ++bc) {

    // Shared array
    int Tile[TILE_DIM][TILE_DIM];

    Parfor (int tr = 0; tr < TILE_DIM; ++tr) {
      for (int tc = 0; tc < TILE_DIM; ++tc) {
        Tile[tr][tc] = A(row, col);
      }
    }

    // ...
  }
}
```

When an inner loop is parallel, tile data should be shared between threads

RAJA provides a LocalArray type to help manage these cases in a portable manner

Local Data

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

TILE_MEM TileArray;
```

The LocalArray type defines a multi-dimensional array of fixed size that can be used in a kernel.

A local array object is allocated inside a kernel

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

TILE_MEM TileArray;

using EXEC_POL = KernelPolicy<
  statement::Tile<1, statement::tile_fixed<TILE_SZ>, loop_exec,
    statement::Tile<0, statement::tile_fixed<TILE_SZ>, loop_exec,

      statement::InitLocalMem<tile_mem_policy, ParamList< # >,
        . . .
        >
    >
  >
>;
```

The local array is allocated for use in a kernel using the 'InitLocalMem' statement. The initialization requires a memory policy and binds the local array object to a slot in the parameter tuple (#).

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

Local Data

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

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

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

Local indices tx, ty are first two entries in param tuple

Lambda args:
- Global indices
- Local tile indices
- LocalArray for tile data

RAJA provides memory policy types for different local array data

`RAJA::cpu_tile_mem` – Use CPU stack memory

`RAJA::cuda_shared_mem` – Use CUDA shared memory (sharable across threads in a CUDA thread block)

`RAJA::cuda_thread_mem` – Use memory local to a CUDA thread

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);
        }

      }
    }
```

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 local arrays as described in the file. 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 array, and one to read from it) because the local array usage must be portable.
- 'ForICount' statements generate local tile indices passed to lambdas in kernel. 'Param' statements identify index args.

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.

Application considerations

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

Apps

- Profile your code to see where performance is most important
  - Do a few kernels dominate runtime?
  - Does no subset of kernels take a significant fraction of runtime?
  - Can you afford to maintain multiple, (highly-optimized) architecture-specific versions of important kernels?
  - Do you require a truly portable, single-source implementation?

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

Apps

- Construct a taxonomy of algorithm patterns/loop structures in your code
  - Is it amenable to grouping into classes of RAJA usage (e.g., execution policies) so that you can propagate changes throughout the code base easily with header file changes?
  - If you have a large code with many kernels, it will be easier to port to RAJA if you define policy types in a header file and apply each to many loops

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

Apps

- 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?
  - Is it important that you limit implementation details to your CS/performance tuning experts?

RAJA promotes flexibility via type parameterization

- Define **type aliases in header files**
  - Easy to explore implementation choices in a large code base
  - Reduces source code disruption

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 **"loop/kernel classes"**
  - Easier to search execution policy parameter space

```
using ELEM_LOOP_POLICY = ...; // in header file

RAJA::forall<ELEM_LOOP_POLICY>( /* do elem stuff */ );
```

Application developers must determine the "loop taxonomy" and policy selection for their code.

Performance portability takes effort

- RAJA (like any programming model) is an enabling technology – not a panacea
  - Achieving and maintaining thread safety in kernels can be challenging
  - Loop characterization and performance tuning are manual processes
    - Good tools are essential!!
  - Memory motion and access patterns are critical. Pay attention to them!
    - True for CPU code as well as GPU code

Performance portability takes effort

- Application coding styles may need to change regardless of programming model (e.g., GPU execution)
  - Change algorithms as needed to ensure correct parallel execution
  - Move variable declarations to innermost scope to avoid threading 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

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

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

= available = in progress = not available (yet)

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

Again, we would appreciate your feedback…

- If you have comments, questions, suggestions, etc., please talk to one of us
- You are welcome to join our Google Group linked to our Github repository home page (https://github.com/LLNL/RAJA)
- Or contact us via our team email list: raja-dev@llnl.gov

Thank you for your attention and participation

Questions?