

A Tutorial Introduction to RAJA

ATPESC19

August 1, 2019



EXASCALE COMPUTING PROJECT

Rich Hornung (hornung1@llnl.gov)

Arturo Vargas (vargas45@llnl.gov)

With contributions from the rest of the RAJA Team



Welcome to the RAJA tutorial

- Today, we will present RAJA, how to use it, and how it enables performance portability
- We will also present background material that will help you think about key issues in parallel computing
- Our discussion will contain both lecture materials and hands-on exercises
- Our main objective is that you learn enough today to start using RAJA in your own code development

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

During the tutorial...

**Please don't hesitate to ask
any question at any time**

We value your feedback...

- If you have comments, questions, suggestions, etc., please let us know
 - Join our Google Group (linked on RAJA Github project home page)
 - Or send email to our project email list: raja-dev@llnl.gov
- We appreciate specific, concrete feedback that helps us improve this tutorial

RAJA and performance portability

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

RAJA provides building blocks that extend the generally-accepted “parallel for” idiom.


RAJA design goals emphasize usability and developer productivity


- Applications should maintain **single-source kernels** (as much as possible)
- **Easy to understand and use** for app developers (most are not CS experts)
- Allow **incremental and selective adoption**
- **Don't 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**


RAJA is developed collaboratively with production application teams.

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

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	HIP
Simple loops	available	available	available	available	available	available	work in progress
Reductions	available	work in progress	available	work in progress	available	available	work in progress
Segments & IndexSets	available	available	available	available	available	available	work in progress
Atomics	available	not available (yet)	available	available	available	available	work in progress
Scans	available	not available (yet)	available	not available (yet)	available	available	work in progress
Complex Loops	available	available	available	work in progress	available	work in progress	work in progress
Layouts & Views	available	available	available	available	available	available	work in progress

 = available

 = work in progress

 = not available (yet)

RAJA is an open source project on Github

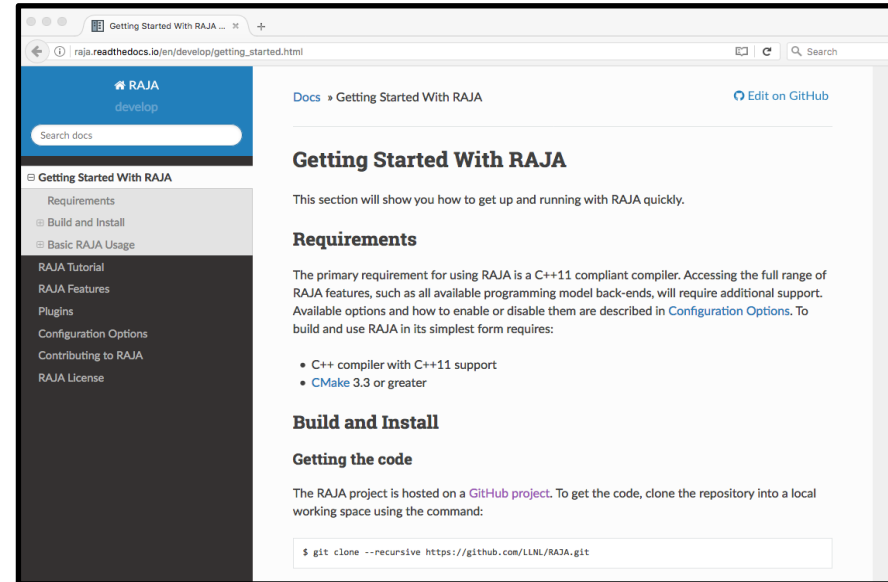
The screenshot shows the GitHub repository page for LLNL/RAJA. The browser address bar displays the URL <https://github.com/LLNL/RAJA>. The repository name is LLNL / RAJA, with 22 Unwatch, 51 Unstar, and 18 Fork actions. The repository is categorized as RAJA Performance Portability Layer, with tags for c-plus-plus, portability, programming-model, parallel-computing, raja, llnl, and cpp. It has 2,557 commits, 30 branches, 13 releases, and 16 contributors. The current branch is develop. Recent commits include a merge pull request #399, an update to the latest BLT master, and updates to cmake, docs, and examples.

Commit	Description	Time Ago
rhornung67 Merge pull request #399 from LLNL/bufix/rhornung67/tut-fixes		2 hours ago
blt @ 949f45a	Update to latest BLT master	4 months ago
cmake	Removed unused cmake variables.	3 months ago
docs	Corrected discussion on nested loop matrix multiplication example.	3 hours ago
examples	Redo example and doc changes to clear warnings/static asserts.	3 days ago

<https://github.com/LLNL/RAJA>

Related projects and other materials...

- **RAJA User Guide:** getting started info, details about today's topics, and more!
- **RAJA Performance Suite:** Collection of loop patterns used to assess compilers and RAJA performance. Used by vendors, DOE platform procurement benchmark, etc.
- **CHAI:** Array abstraction library that automatically migrates data as needed based on RAJA execution contexts



These are linked on the RAJA Github project.

You will need to do some simple preparation before attempting the tutorial exercises

- We've set up two options for you:
 - Docker container for your laptop (Mac only)
 - ALCF machines (Cooley and Theta)
 - Get the RAJA code, and put it someplace in your home directory on those machines:
 - `git clone --recursive https://github.com/LLNL/RAJA.git`
 - `cd RAJA`
 - `git checkout ATPESC2019` (a branch we have set up for this tutorial)

Please try to do this before we get to the exercises.

If you will use the Docker container...

- You will need to install Docker Desktop (or similar)
 - See <https://docs.docker.com/install> if you need to do this
 - Create account if needed. Login. Download image for your OS. Install on your machine.
- Get the RAJA tutorial Docker container
 - Run the following command:
docker run -it rajaorg/raja-tutorial:atpesc19
 - This puts you into a bash terminal inside the Docker container, which already has RAJA installed

Please try to do this before we get to the exercises.

Before we dig into RAJA, we will discuss some things that are helpful to keep in mind during the tutorial...

Why are we interested in parallel computing?

- We hope that when we can **perform multiple operations simultaneously**, our applications will run faster

Parallel computing comes in various forms

- *What are some forms of parallel computing?*

Parallel computing comes in various forms

- *What are some forms of parallel computing?*
 - **Instruction-level Parallelism (ILP)** – multiple machine instructions run at the same time
 - Typically, the result of compiler optimizations for specific hardware
 - Instruction pipelining
 - Out-of-order execution
 - Branch prediction
 - Etc.

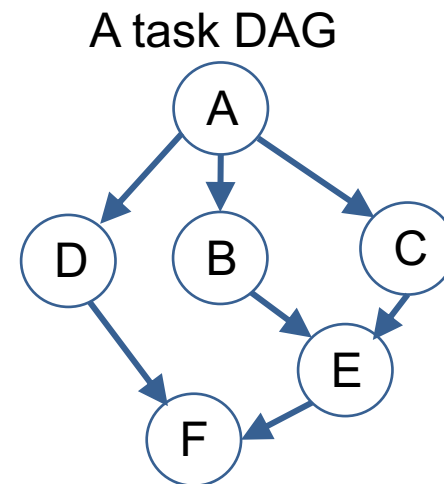
$$\begin{array}{l} \mathbf{a} = \mathbf{b} + \mathbf{c} \\ \mathbf{d} = \mathbf{e} * \mathbf{f} \\ \mathbf{g} = \mathbf{a} + \mathbf{d} \end{array} \left. \vphantom{\begin{array}{l} \mathbf{a} = \mathbf{b} + \mathbf{c} \\ \mathbf{d} = \mathbf{e} * \mathbf{f} \\ \mathbf{g} = \mathbf{a} + \mathbf{d} \end{array}} \right\} \text{Independent operations}$$

The amount of available parallelism depends on how many operations can be performed simultaneously

Parallel computing comes in various forms

- *What are some forms of parallel computing?*
 - Instruction-level Parallelism (ILP) – multiple machine instructions run at the same time
 - **Task (functional) parallelism** – tasks run in parallel using same or different data

The amount of available parallelism depends on the number of independent tasks



Parallel computing comes in various forms

- *What are some forms of parallel computing?*
 - Instruction-level Parallelism (ILP) – multiple machine instructions run at the same time
 - **Task (functional) parallelism** – tasks run in parallel using same or different data
 - **Data Parallelism** – same operation is applied to different subsets of data; e.g., SIMD vectorization

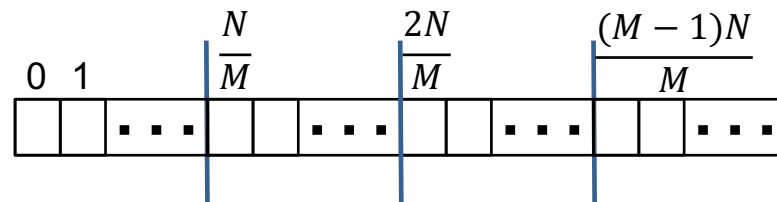
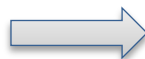
The amount of available parallelism is proportional to size of input data

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

Parallel computing comes in various forms

- *What are some forms of parallel computing?*
 - Instruction-level Parallelism (ILP) – multiple machine instructions run at the same time
 - **Task (functional) parallelism** – tasks run in parallel using same or different data
 - **Data Parallelism** – same operation is applied to different subsets of data

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



If this loop runs in T time units on one process/thread, we hope it will run in T / M time units in parallel on M processors/threads ($M \leq N$)

This tutorial focuses on “fine-grained” (loop-level) data parallelism.

Data dependencies are a key inhibitor of parallelism

- 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 main inhibitor of parallelism

- 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

What's the issue?

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

Data dependencies are a main inhibitor of parallelism

- 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

What's the issue?

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

Each loop iteration writes to 'sum'

We'll discuss RAJA reductions later in the tutorial.

Data dependencies are a main inhibitor of parallelism

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

What's the issue?

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

Data dependencies are a main inhibitor of parallelism

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

What's the issue?

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

$x[i-1]$ 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

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

What's the issue?

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


Data dependencies are a main inhibitor of parallelism

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

What's the issue?

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

$A(c, r)$ and $A(r, c)$ depend on each other

Data dependencies are a main inhibitor of parallelism

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

What's the issue?

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

Data dependencies are a main inhibitor of parallelism

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

What's the issue?

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

There is no issue. All loop iterations are independent.

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

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

- Amdahl's law:

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

$S(n)$ is the theoretical maximum speedup of a fixed 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 maximum theoretical “speedup” we can achieve in a parallel program

- Amdahl's law

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

$S(n)$ is the theoretical maximum speedup of a fixed 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 can increase as we run on more processors. But, for a fixed workload, we cannot continue to add processors and expect additional speedup.

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

- Amdahl's law

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

$S(n)$ is the theoretical maximum speedup of a fixed 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 can increase as we run on more processors. But, for a fixed workload, we cannot continue to add processors and expect additional speedup.

If only 50% of an application can run in parallel ($p = 0.5$), what does Amdahl's law tell us the maximum speedup we could observe on any number of processors?

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

- Amdahl's law

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

$S(n)$ is the theoretical maximum speedup of a fixed 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 can increase as we run on more processors. But, for a fixed workload, we cannot continue to add processors and expect additional speedup.

If only 50% of an application can run in parallel ($p = 0.5$), what does Amdahl's law tell us the maximum speedup we could observe on any number of processors? **2x**

Try plugging other values of p , between 0 and 1, into the formula.

Theoretical speedup is always limited by sequential portions of your code

- Amdahl's law

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

$S(n)$ is the theoretical maximum speedup of a fixed 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:

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

$$\lim_{n \rightarrow \infty} S(n) = \frac{1}{(1 - p)}$$

How do we measure what we actually gain from parallelism?

- 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”

How do we measure what we actually gain from parallelism?

- Compare sequential run time and parallel run time

$$S_n = n (E_n = 1) \longrightarrow \text{“Perfect (Ideal) scaling”}$$

In reality, $S_n < n$ ($E_n < 1$) most of the time!

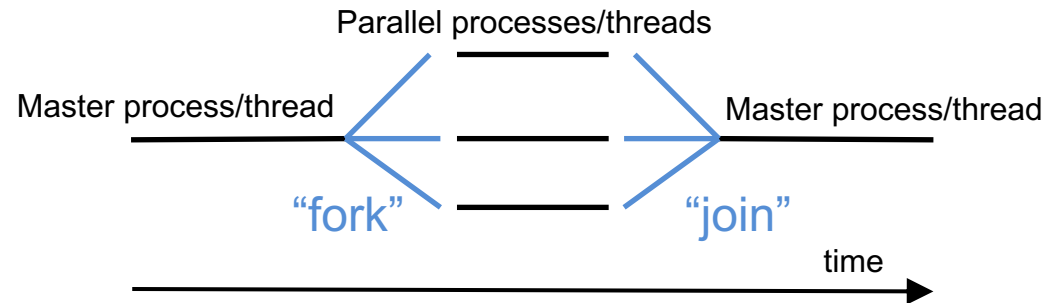
How do we measure how much we gain from parallelism?

- Compare sequential run time and parallel run time

$$S_n = n (E_n = 1) \longrightarrow \text{“Perfect (Ideal) scaling”}$$

In reality, why is $S_n < n$ ($E_n < 1$) most of the time?

- **Synchronization overhead**



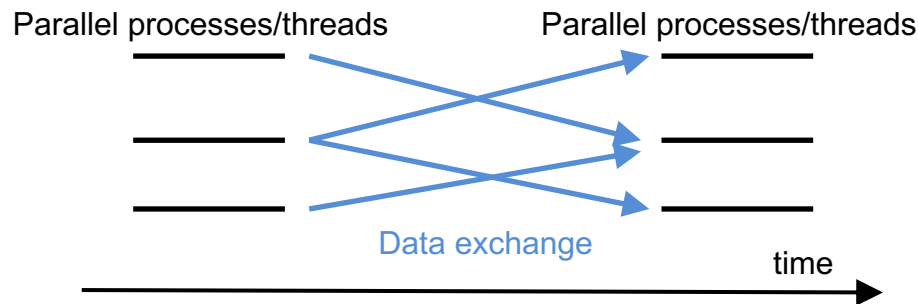
How do we measure how much we gain from parallelism?

- Compare sequential run time and parallel run time

$$S_n = n (E_n = 1) \longrightarrow \text{“Perfect (Ideal) scaling”}$$

In reality, why is $S_n < n$ ($E_n < 1$) most of the time?

- Synchronization overhead
- **Communication overhead**



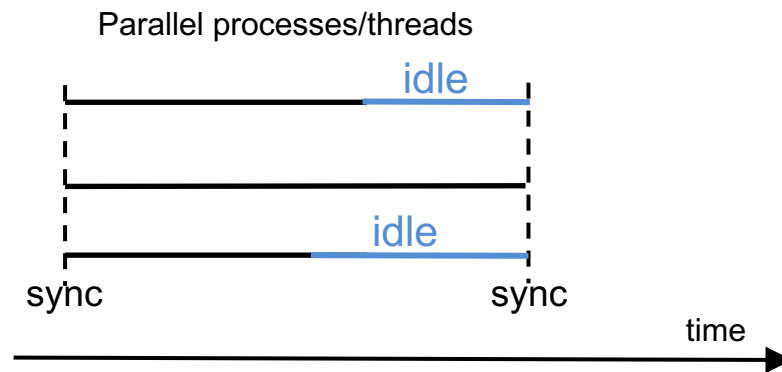
How do we measure how much we gain from parallelism?

- Compare sequential run time and parallel run time

$$S_n = n (E_n = 1) \longrightarrow \text{“Perfect (Ideal) scaling”}$$

In reality, why is $S_n < n$ ($E_n < 1$) most of the time?

- Synchronization overhead
- Communication overhead
- **Load imbalance**



How do we measure how much we gain from parallelism?

- Compare sequential run time and parallel run time

$$S_n = n \quad (E_n = 1) \quad \longrightarrow \quad \text{“Perfect (Ideal) scaling”}$$

In reality, why is $S_n < n$ ($E_n < 1$) most of the time?

- Synchronization overhead
- Communication overhead
- Load imbalance
- **Many algorithms contain sections that do not benefit from parallelization; e.g., parts that are inherently serial** (remember Amdahl's law)

Something to ponder....

- Recall definition of parallel speedup

$$S_n = T_1 / T_n$$

Theoretically, it is not possible for $S_n > n$ because of Amdahl's Law.

In practice, it is possible to observe $S_n > n$. This is called “superlinear speedup”.

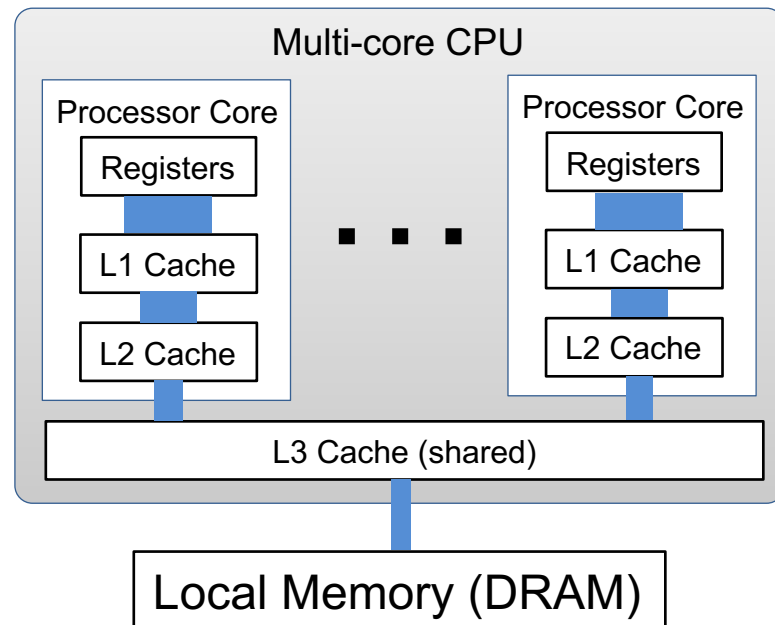
How can this happen?

- A useful reference on superlinear speedup:
 - “Superlinear Speedup in HPC Systems: Why and When?” by S. Ristov et al.
(<https://ieeexplore.ieee.org/document/7733347>)

Understanding basic architecture features helps to program for good performance

Good to know

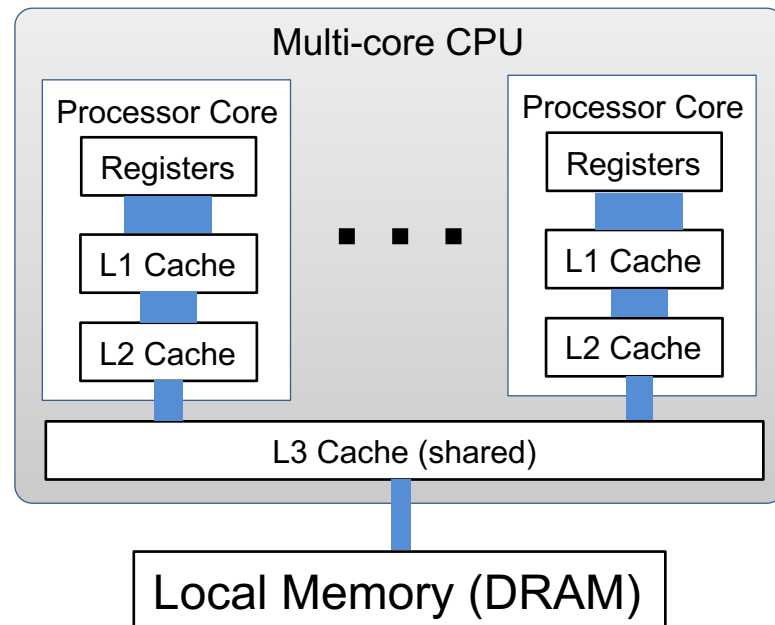
- Modern multi-core CPUs have a hierarchy of memory levels
 - Some are local to each core: registers, caches
 - Some memory is shared with other cores or CPUs: caches, node local memory



Understanding basic architecture features helps to program for good performance

Good to know

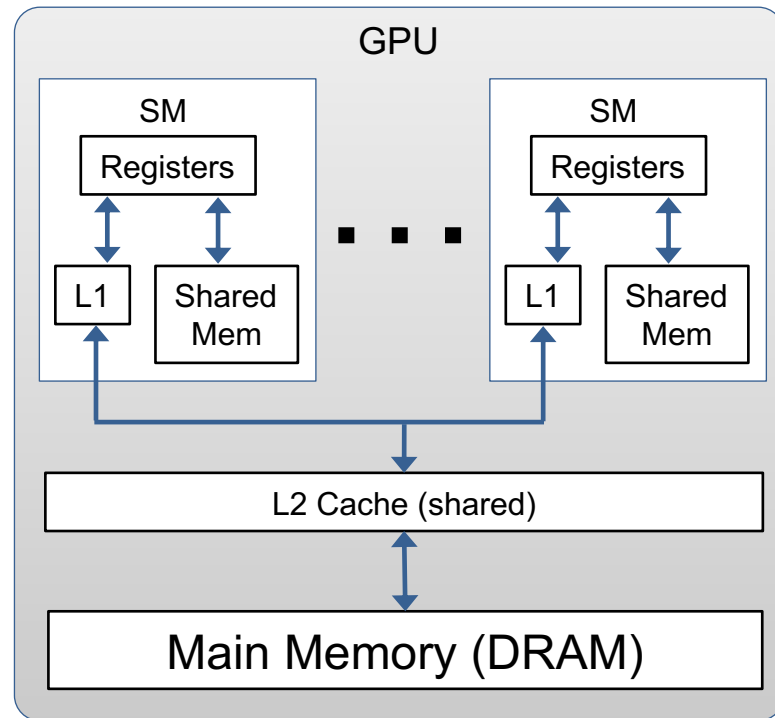
- Modern multi-core CPUs have a hierarchy of memory levels
 - Some are local to each core: registers, caches
 - Some memory is shared with other cores or CPUs: caches, node local memory
- Data move through the memory hierarchy to each processor core as they are used and migrate away when not used
- Memory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



Understanding basic architecture features helps to program for good performance

Good to know

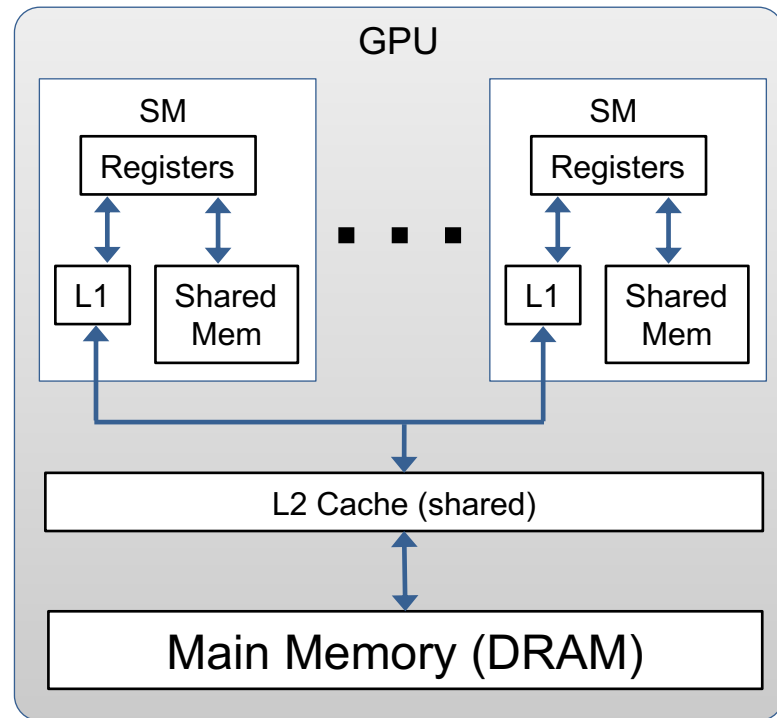
- GPU has multiple streaming multiprocessors (SMs) and a memory hierarchy
 - Some memory levels are local to each SM, some are shared by SMs



Understanding basic architecture features helps to program for good performance

Good to know

- GPUs have multiple streaming multiprocessors (SMs) and a memory hierarchy
 - Some levels are local to each SM, some are shared by SMs
- Each SM has a “large” register file, an L1 cache and 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
 - Supports fast atomic memory operations
- 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: (cache) **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 (cache) **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 a single 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>)

Before we dig into RAJA....

...a few other things to mention

RAJA makes heavy use of C++ templates

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

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

RAJA makes heavy use of C++ templates

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

- Here, “ExecPol”, “IdxType”, “LoopBody” are C++ types you specify at **compile-time**

Like this...

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

- “IdxType” and “LoopBody” types are deduced by the compiler based on what you specify

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

Like this...

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

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

Users pass loop bodies to RAJA as C++ lambda expressions (C++11)

```
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}
```

Users pass loop bodies to RAJA as C++ lambda expressions (C++11)

```
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
 - Value or reference ([=] vs. [&])? By-value is required for GPU execution, when using RAJA reductions, etc.
 - **We recommend using capture by-value** in all cases, as shown above

Users pass loop bodies to RAJA as C++ lambda expressions (C++11)

```
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
 - Value or reference ([=] vs. [&])? By-value is required for GPU execution, when using RAJA reductions, etc.
 - **We recommend using capture by-value** in all cases, as shown above
- The **parameter list** are arguments passed to lambda function body; e.g., (**int i**) is “loop variable”

Users pass loop bodies to RAJA as C++ lambda expressions (C++11)

```
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
 - Value or reference ([=] vs. [&])? By-value is required for GPU execution, when using RAJA reductions, etc.
 - **We recommend using capture by-value** in all cases, as shown above
- The **parameter list** are arguments passed to lambda function body; e.g., (**int i**) is “loop variable”
- A lambda used in a CUDA kernel requires a *device annotation*: [=] **__device__** (...) { ... }

Users pass loop bodies to RAJA as C++ lambda expressions (C++11)

```
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
 - Value or reference ([=] vs. [&])? By-value is required for GPU execution, when using RAJA reductions, etc.
 - **We recommend using capture by-value** in all cases, as shown above
- The **parameter list** are arguments passed to lambda function body; e.g., (**int i**) is “loop variable”
- A lambda used in a CUDA kernel requires a *device annotation*: [=] **__device__** (...) { ... }

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

“Bring your own” memory management

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

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

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

“Bring your own” memory management

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

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

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

- 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

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

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

For simplicity, all tutorial exercises use unified memory

Let's start simple...

Simple loop execution

A typical for-loop written in C/C++ exposes all aspects of execution explicitly

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

C-style for-loop

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

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

RAJA encapsulates loop execution details

C-style for-loop

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

RAJA-style loop

```
using EXEC_POL = ...;

RAJA::RangeSegment range(0, N);

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

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

RAJA encapsulates loop execution details

C-style for-loop

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

RAJA-style loop

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

Typically, these definitions go in a header file.

Learn to love using the C++ “using” directive.

RAJA encapsulates loop execution details

C-style for-loop

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

RAJA-style loop

```
using EXEC_POL = ...;

RAJA::RangeSegment range(0, N);

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

Same loop body.

With RAJA, the loop header is different, but the loop body is the same (in most cases).

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** (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 passed as a C++ lambda expression**
 - Lambda argument is the loop variable

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

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

```
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
```

Examples of RAJA loop execution policy types.

RAJA provides a variety of execution policy types...

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

Implementations for some policies are **works-in-progress.

RAJA support for simple loops

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

■ = available

■ = work in progress

■ = not available (yet)

Reductions

Reduction is a common and important parallel pattern

dot product: $\text{dot} = (a, b)$, where a and b are vectors and dot is a scalar

C-style

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

Reduction is a common and important parallel pattern

C-style

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

What might a parallel implementation look like?

Reduction is a common and important parallel pattern

C-style

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

What might a parallel implementation look like?

Suppose $N = 8$ and we have $P = 4$ processors

Reduction is a common and important parallel pattern

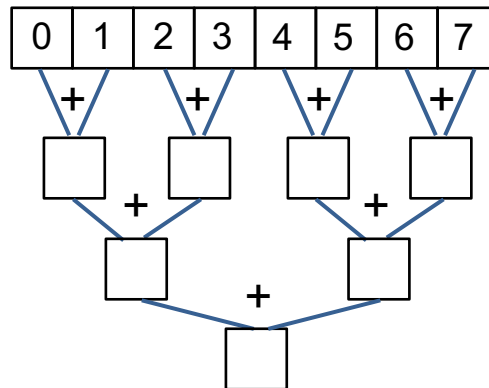
C-style

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

What might a parallel implementation look like?

Suppose $N = 8$ and we have $P = 4$ processors

“Tree-based” algorithm



4 adds in parallel
 2 adds in parallel
 1 add
 } $\log_2(8) = 3$ steps

Reduction is a common and important parallel pattern

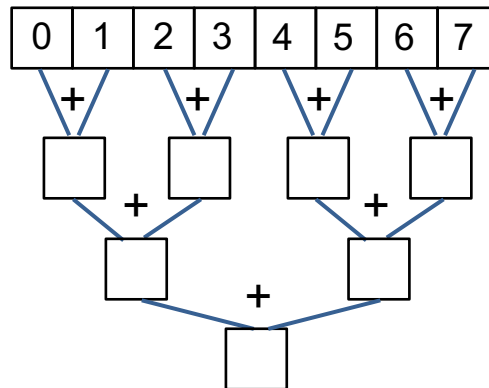
C-style

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

“Extra credit”:

What does Brent’s Theorem tell us about this algorithm?

“Tree-based” algorithm



4 adds in parallel
 2 adds in parallel
 1 add
 } $\log_2(8) = 3$ steps

Reduction is a common and important parallel pattern

C-style

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

“Extra credit”:

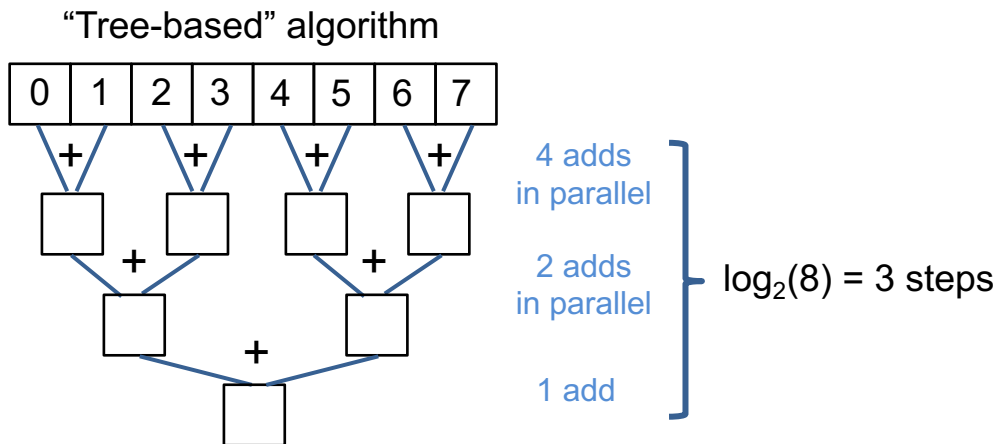
What does Brent’s Theorem tell us about this algorithm?

$$T_p \leq \frac{N}{p} + \log_2(N)$$

Reference:

“Distributed Algorithms and Optimization” by Reza Zadeh

(https://stanford.edu/~rezab/dao/notes/lecture01/cme323_lec1.pdf)



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();
```

- 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();
```

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)
- **After loop runs, get reduced value via 'get' method**

Elements of RAJA reductions...

```
RAJA::ReduceSum< REDUCE_POLICY, DTYPE > sum(init_val);
```

```
RAJA::forall< EXEC_POLICY >(... {  
    sum += func(i);  
});
```

```
type reduced_sum = sum.get();
```

The reduction policy and loop execution policy **must be compatible**.

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;
```

Examples of 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 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 >( a_range, [=](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:

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

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

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

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

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

- *What are the reduced values?*
 - Sum = -9
 - Min = -10
 - Max = 10
 - Max-loc = $N/2$
 - Min-loc = $N/2 - 1$ or $N/2 + 1$ (order-dependent)

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

RAJA support for reductions

	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)

Hands on Exercises

Preparation for the hands-on exercises

- We've set up two options for you:
 - Docker container for your laptop (Mac only)
 - ALCF machines (Cooley and Theta)
 - Get the RAJA code, and put it someplace in your home directory on those machines:
 - `git clone --recursive https://github.com/LLNL/RAJA.git`
 - `cd RAJA`
 - `git checkout ATPESC2019` (a branch we have set up for this tutorial)

If you will use the Docker container...

- You will need to install Docker Desktop (or similar)
 - See <https://docs.docker.com/install> if you need to do this
 - Create account if needed. Login. Download image for your OS. Install on your machine.
- Get the RAJA tutorial Docker container
 - Run the following command:
docker run -it rajaorg/raja-tutorial:atpesc19
 - This puts you into a bash terminal inside the Docker container, which already has RAJA installed

If you're using ALCF machines...

- We've set up build scripts for you to configure and build the code
 - Scripts live in the directory RAJA/scripts/alcf-builds
 - Each script has instructions to get on a compute node and set up your basic environment
 - Before running a build script, run the commands specified in the build script first
 - Then, in the top-level RAJA directory, build the RAJA code:
 - Run script for machine and compiler you want to use; e.g.,
`./scripts/alcf-builds/cooley_nvcc9.1_clang4.0.sh`
 - `cd` into the build directory created by the script
 - Type `'make -j'` to build RAJA and `'make test'` to check if it works

How to work through the exercises...

- In either case, each exercise involves:
 - Editing the exercise source file to insert RAJA code
 - Recompiling the code (i.e., run make in the build directory)
 - Running the exercise executable file (i.e., enter executable name in 'bin' directory)
 - Checking the output to see if what you did passes or fails the checks
- Each exercise source file contains a description of the exercise and the RAJA features you will use to perform the exercise
- The locations to modify in the exercise source files are indicated by comments containing the text 'TO DO...' and 'EXERCISE'

Exercise #1: vector addition

- See file: **RAJA/exercises/tutorial_halfday/ex1_vector-addition.cpp**
 - It contains C-style sequential and OpenMP implementations of 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 sequential and OpenMP variants using RAJA::forall() methods and execution policies (also do the same for CUDA if you can). Run the code and check your results. The file has empty code sections indicated with comments for you to fill in and methods you can use to check your work and print results.*

See <https://raja.readthedocs.io/en/v0.9.0/feature/policies.html> for a listing of RAJA loop execution policies.

Exercise #1 solution

- Your code should look something like this:

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

Where the execution policy type is chosen for each case (seq, OpenMP, CUDA).

- The file **RAJA/exercises/tutorial_halfday/ex1_vector-addition_solution.cpp** contains complete implementations of the solution to exercise #1. It also shows multiple “sequential” variants using different execution policies.

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

Exercise #2: approximate pi

- Recall some basic calculus:

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

- See file: **RAJA/exercises/tutorial_halfday/ex2_approx-pi.cpp**
 - It contains C-style sequential and OpenMP loops that use this formula to approximate pi using *Riemann integration*.
- Exercise: Implement RAJA sequential and OpenMP variants of the pi approximation using RAJA::forall() methods and RAJA reductions (also do the same for CUDA if you can). The file contains empty code sections indicated with comments for you to fill in and methods you can use to check your work and print results.*

See <https://raja.readthedocs.io/en/v0.9.0/feature/policies.html> for a listing of RAJA loop execution and reduction policies.

Exercise #2 solution

- Your code should look something like this:

```
RAJA::ReduceSum< REDUCE_POL, double > pi(0.0);

RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N), [=] (int i) {
    double x = (double(i) + 0.5) * dx;
    pi += dx / (1.0 + x * x);
});
```

Where the execution policy type is provided in the file for each case and you have filled in a compatible reduction policy type.

- The file RAJA/exercises/tutorial_halfday/ex2approx-pi_solution.cpp contains complete implementations of the solution to exercise #2.

Iteration spaces : Segments and IndexSets

A RAJA “Segment” is defines a loop iteration space

- A “Segment” defines a set of loop indices to run as a unit

Contiguous range [beg, end)



Strided range [beg, end, stride)




List of indices (indirection)




A RAJA “Segment” is the basic means to define a loop iteration space

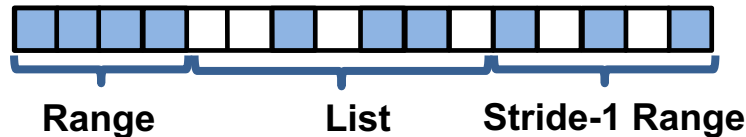
- A “Segment” defines a set of loop indices to run as a unit

Contiguous range [beg, end) 

Strided range [beg, end, stride) 

List of indices (indirection) 

- An “Index Set” is a container of segments (of arbitrary types)



You can run all segments in an IndexSet in one RAJA loop execution template.

A RangeSegment defines a contiguous sequence of indices (stride-1)

```
RAJA::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 srangel( 0, N, 2 );
```

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

Runs loop indices: {0, 2, 4, ...}

A RangeStrideSegment defines a strided sequence of indices

```
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}

RAJA supports negative indices and strides.

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

```
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

```
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 may contain multiple 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);
```

A RAJA IndexSet is a container of Segments

```
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 );
```

A RAJA IndexSet is a container of Segments

```
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

A RAJA IndexSet is a container of Segments

```
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);
```

Segment types
must be specified
at compile time

```
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 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.

RAJA Segments and IndexSets work with all back-ends

Iteration spaces

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



= available



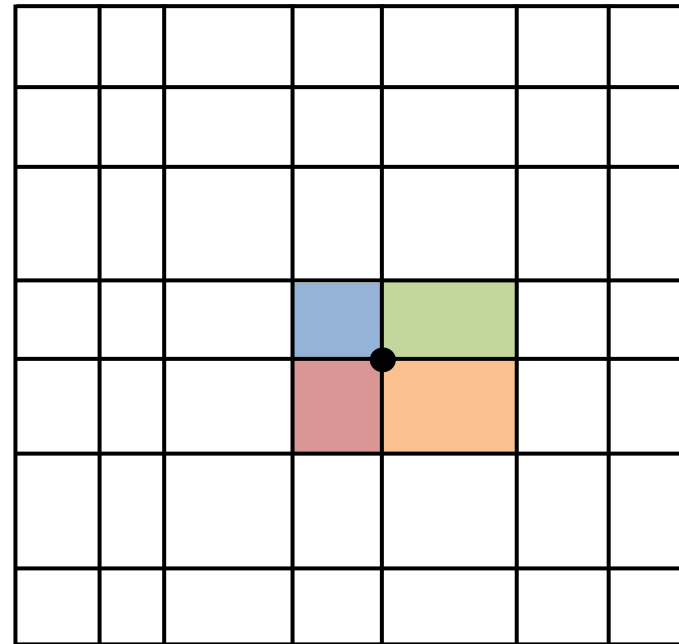
= in progress



= not available (yet)

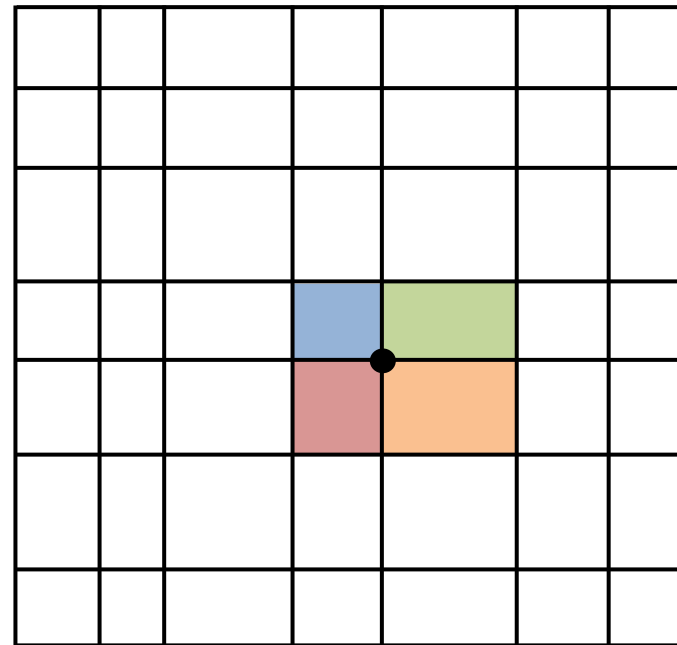
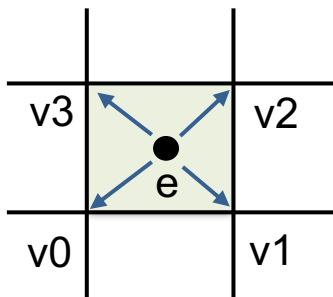
IndexSets can help enable parallelism

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



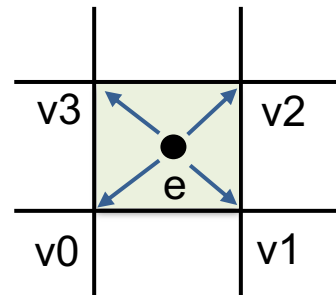
IndexSets can help enable parallelism

- Consider an irregularly-spaced 2D Cartesian mesh
- At each mesh vertex, we want to compute the average area of the 4 surrounding elements
 - For each element e , add $\frac{1}{4} \text{area}(e)$ to $\text{area}(v_i)$, $i = 0, \dots, 3$



A C-style serial code for the vertex area

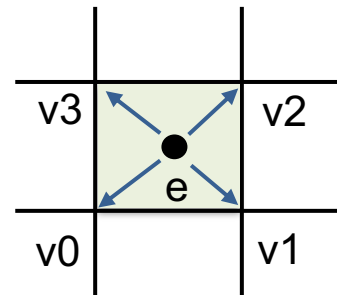
```
for (int ie = 0 ; ie < N_elem ; ++ie) {  
  
    int* iv = e2v_map(ie);  
  
    areav[ iv[0] ] += areae[ie] / 4.0 ;  
    areav[ iv[1] ] += areae[ie] / 4.0 ;  
    areav[ iv[2] ] += areae[ie] / 4.0 ;  
    areav[ iv[3] ] += areae[ie] / 4.0 ;  
  
}
```



As written, will this code work 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 work in parallel?

No. There is a data race at each vertex.

One approach: partition the elements into four subsets 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 ;

}

```

0	1	0	1	0
2	3	2	3	2
0	1	0	1	0
2	3	2	3	2
0	1	0	1	0

No two elements with same color share a vertex

One approach: partition the elements into four subsets 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 ;  
  
}
```

0	1	0	1	0
2	3	2	3	2
0	1	0	1	0
2	3	2	3	2
0	1	0	1	0

Will the results be reproducible?

One approach: partition the elements into four subsets 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 ;

}

```

0	1	0	1	0
2	3	2	3	2
0	1	0	1	0
2	3	2	3	2
0	1	0	1	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

- See file: **RAJA/exercises /tutorial_halfday/ex3_colored-indexset.cpp**
 - It contains C-style sequential and OpenMP variants of the vertex area calculation described on the previous slide. They use arrays that enumerate the elements of each color.

- *Exercise: Implement and run a RAJA OpenMP variant of the vertex area calculation that uses a RAJA IndexSet containing 4 ListSegments and one call to a RAJA::forall() method (do the same for CUDA if you can). The file contains the 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.*

Exercise #3 solution

- Your code should look like the following, where you have filled in the appropriate segment type and IndexSet execution policy:

```

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()) );

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 ;
});

```

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

Atomic operations

RAJA provides portable atomic operations

```
int* x = ...
```

```
int* y = ...
```

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

```
    RAJA::atomicAdd< ATOMIC_POLICY >(&x[i], 1);
```

```
    RAJA::atomicSub< ATOMIC_POLICY >(&y[i], 1);
```

```
} );
```

Atomic operations perform updates at specific memory addresses (write or read-modify-write) where only one thread or process at a time can do the update.

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.

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 easier portability.

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

- “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` style interface (RAJA::AtomicRef)

The RAJA User Guide describes these atomic operations in detail.

RAJA support for atomics

	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)

Exercise #4: atomic histogram

- You have an integer array of length N , whose element values are in the set $\{0, 1, 2, \dots, M-1\}$, where $M < N$. You want to build a *histogram* array of length M such that the i -th entry in the array is the number of occurrences of the value i in the original array.
- See file: **RAJA/exercises/tutorial_halfday/ex4_atomic-histogram.cpp**
 - It contains C-style sequential and OpenMP implementations of the histogram calculation.
- *Exercise: Implement and run RAJA sequential and OpenMP variants of loops to compute the histogram array using RAJA::forall methods and RAJA atomic operations (do the same for CUDA if you can). The file contains empty code sections for you to fill in and methods you can use to check your work and print results.*

See <https://raja.readthedocs.io/en/v0.9.0/feature/policies.html> for a listing of RAJA loop execution and atomic policies.

Exercise #4 solution

- Your code should look something like this:

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

Where the atomic policy type is compatible with the execution policy for each case (seq, OpenMP, CUDA).

- The file **RAJA/exercises/tutorial_halfday/ex4_atomic-histogram_solution.cpp** contains complete implementations of the solution to exercise #4. It also shows variants that use the RAJA `auto_atomic` policy.

Scan operations

Scan is an important building block for parallel algorithms

- It is a key primitive to convert serial operations to parallel implementations
 - Based on reduction tree and reverse reduction tree
 - 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>)

Parallel prefix sum is the most common scan

```
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 “+”.

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 result in 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.

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

RAJA support for scans

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



= available



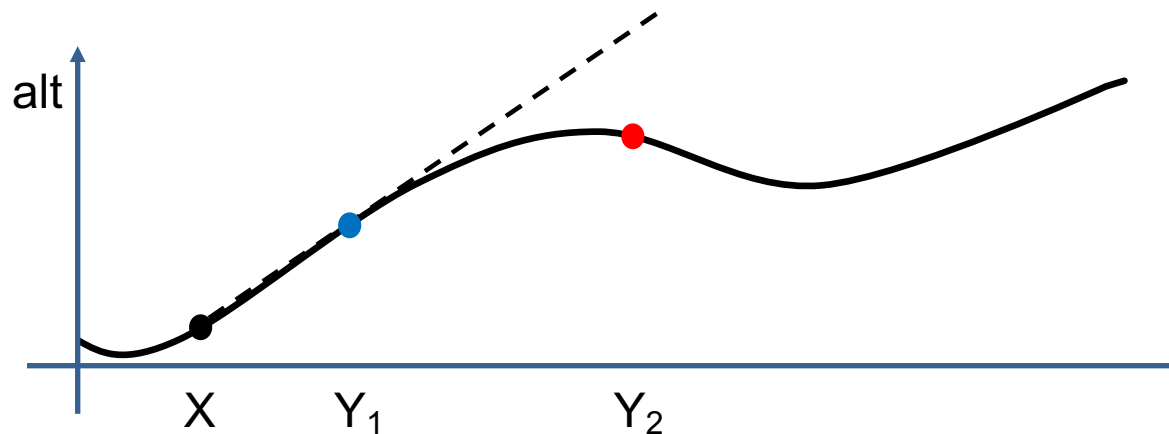
= in progress



= not available (yet)

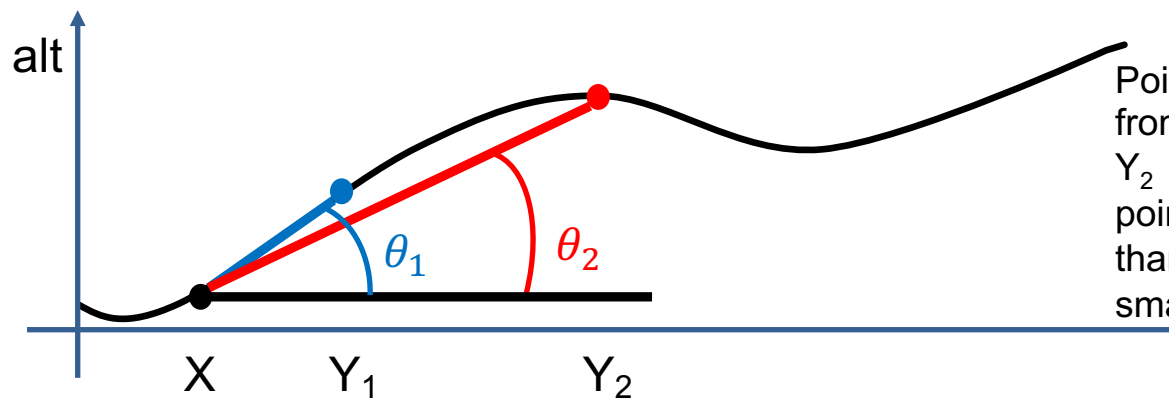
Exercise #5: the line-of-sight problem

- 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

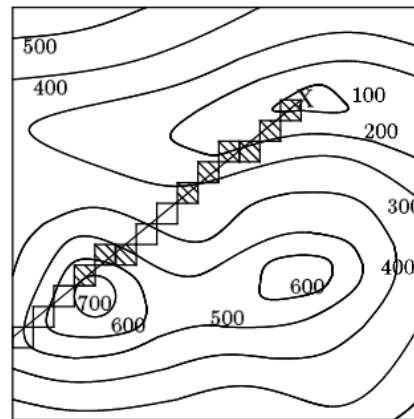
- The *line-of-sight* problem: given an observation point at X on a terrain map, and a set of points along a ray starting at X, find which points on the terrain are visible from X.
- A point at Y on the ray is visible from the point at X if and only if no other point on the terrain between the points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



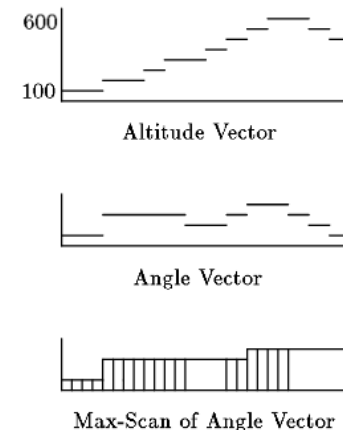
Point at Y₁ (blue) can be seen from the point at X and point at Y₂ (red) cannot. Although the point at Y₂ has a higher altitude than the point at Y₁, it has a smaller vertical angle.

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 a vector 'alt' be defined so that alt[i] is the altitude at point Y_i.
- Let the vector 'dist' be defined so that dist[i] is the horizontal distance between point X and point Y_i.
- We compute an angle vector 'ang' that holds the vertical angle at each point computed as:
 - $\text{ang}[i] = \tan^{-1}((\text{alt}[i] - \text{altX})/\text{dist}[i])$
- A *max scan* on the angle vector tells us if the point Y_i is visible from X:
 - If $\text{ang}[i] \geq \text{ang_max}[i]$, then Y_i is visible, 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

- See file: **RAJA/exercises/ex5_line-of-sight.cpp**
 - It contains a C-style sequential code that implements the line-of-sight algorithm on the previous slide.
- *Exercise: Implement and run sequential and OpenMP 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 (do the same for CUDA if you can). The file contains empty code sections for you to fill in and methods you can use to check your work and print results.*

See <https://raja.readthedocs.io/en/v0.9.0/feature/scan.html> for a listing of RAJA scan execution policies and operators.

Exercise #5 solution

- Your code should look something like this (where you have filled in the execution policy):

```
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;  
    }  
});
```

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

Views and Layouts

Matrices and tensors are ubiquitous in scientific computing

- 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 patterns

- A RAJA View wraps a pointer to enable indexing following a 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 patterns

- A RAJA View wraps a pointer to enable indexing that follows a 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 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

```
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 i = 0; i < Nn; ++i) {
    Aview(i0, i1, ..., i) = ...;
}
```

Stride-1 data access

```
// iterate over jth index and hold others fixed
```

```
for (int j = 0; j < Nj; ++j) {
    Aview(i0, i1, ..., j, ..., iN) = ...;
}
```

Data access stride is
 $N_n * \dots * N_{(j+1)}$

The right-most index is stride-1 using the default Layout<DIM>.

RAJA provides methods to make layouts for other indexing patterns

- A “permutated layout” changes the striding order

```
std::array<RAJA::idx_t, 3> perm {{1, 2, 0}};
```

```
RAJA::Layout< 3 > perm_layout =  
RAJA::make_permuted_layout( {{5, 7, 11}}, perm);
```

This gives a 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)

A permuted layout changes the striding order

```
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, int> > 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)

A default layout uses the “identity” permutation (i.e., {0, 1, 2}).

An offset layout applies offsets 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 layout 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

- All Layout objects have a permutation. So there is no ``RAJA::PermutedLayout`` type. For example,

```
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. For example,

```
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 $[-1, 2] \times [-5, 5]$.

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 $[-1, 2] \times [-5, 5]$.

- *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 $[-1, 2] \times [-5, 5]$.

- *Which index is stride-1?*

Index '1' (**right-most**) is stride-1 (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 $[-1, 2] \times [-5, 5]$.

- *Which index is stride-1?*

Index '1' (**right-most**) is stride-1 (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 $[-1, 2] \times [-5, 5]$.

- *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 $[-1, 2] \times [-5, 5]$ (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 $[-1, 2] \times [-5, 5]$ (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 $[-1, 2] \times [-5, 5]$ (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to the 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 $[-1, 2] \times [-5, 5]$ (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to the 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 $[-1, 2] \times [-5, 5]$ (same as previous example).

- *Which index is stride-1?*

Index '0' has stride-1 (due to the 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

```
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

```
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

```
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”?

$$\text{lin} = 188 (= 1 + 3 * 11 + 2 * 11 * 7)$$

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 support for views and layouts

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



= available



= in progress

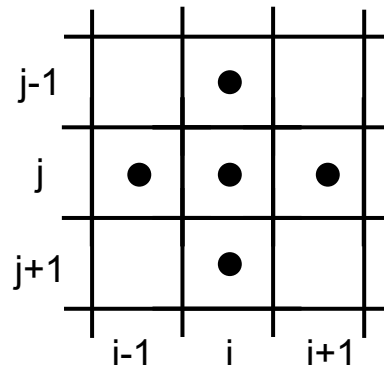


= not available (yet)

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” matrix has an entry for each element on the mesh interior:

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

and the “B” matrix has an entry each element on the mesh interior plus a “halo” layer 1 element wide around the interior:

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

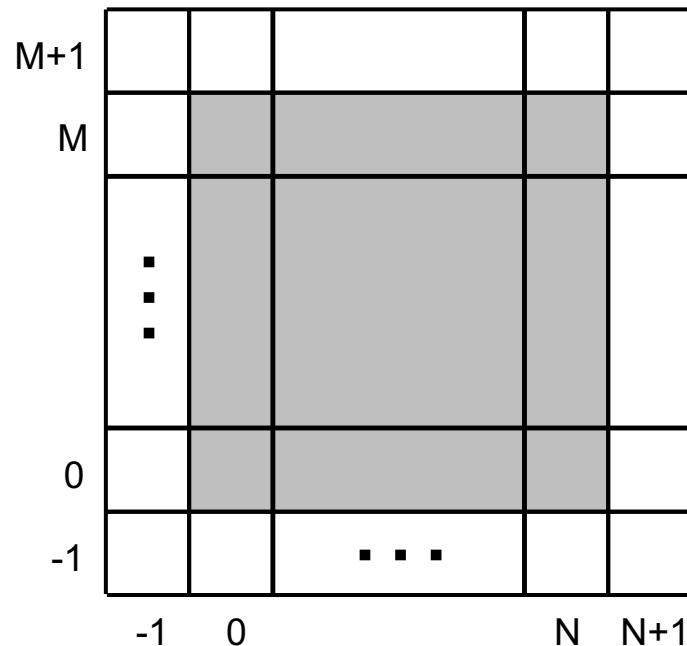
Exercise #6: 5-point stencil

- 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) and use RAJA Views to write the loop body “naturally” as:

$$A_{\text{view}}(i, j) = B_{\text{view}}(i, j) + B_{\text{view}}(i-1, j) + B_{\text{view}}(i+1, j) + B_{\text{view}}(i, j-1) + B_{\text{view}}(i, j+1)$$

- That is, so it looks like this:

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



Exercise #6: 5-point stencil

- The file **RAJA/exercises/ex6_stencil-offset-layout.cpp** contains C-style sequential implementations of the 5-point stencil computation.
 - Part A: Assumes that the column (j-loop) indexing is stride-1.
 - Part B: Assumes that the row (i-loop) indexing is stride-1.
 - Note that the manual index offset arithmetic in the inner loop bodies is different!
- *Exercise: Implement and run sequential variants of parts A and B using RAJA Views. Note that you are to fill in empty code sections inside C-style for-loops. The goal of this exercise is for you to learn the mechanics of creating and using RAJA Layouts and Views. The file contains methods you can use to check your work and print results.*
- Note: because you are using RAJA Views, the loop bodies look the same in each case; like this:

$$\begin{aligned} \text{Aview}(i, j) = & \text{Bview}(i, j) + \text{Bview}(i-1, j) + \text{Bview}(i+1, j) + \\ & \text{Bview}(i, j-1) + \text{Bview}(i, j+1) \end{aligned}$$

Exercise #6 solution

- For part B, your construction of RAJA Views should look like this:

```
std::array<RAJA::idx_t, DIM> perm {{1, 0}}; // 'i' index (position zero)
                                           // is stride-1

RAJA::OffsetLayout<DIM> pB_layout =
    RAJA::make_permuted_offset_layout( {{-1, -1}}, {{Nc_tot-2, Nr_tot-2}},
                                       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 file `RAJA/exercises/ex6_stencil-offset-layout_solution.cpp` contains complete implementations of the solution to parts A and B of exercise #6.

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

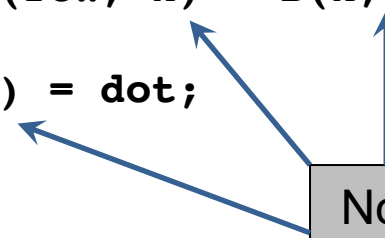
```
for (int row = 0; row < N; ++row) {
    for (int col = 0; col < N; ++col) {

        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;
    }
}
```

C-style
nested
loops

For a RAJA implementation, we could use nested 'forall' statements...

```
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 in this example to simplify multi-dimensional indexing.

...but, 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
 - 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 same as before.

The RAJA::kernel interface has four basic concepts

- 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

```

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 ) {
    // ...
} );

```

statement::For<1, exec_policy_row,
statement::For<0, exec_policy_col,
statement::Lambda<0>

'0' → col
'1' → row

Integer template 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

```
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-loop order in a C-style implementation.

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

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

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

```
Lambda<0>
```

```
CudaSyncThreads
```

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

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

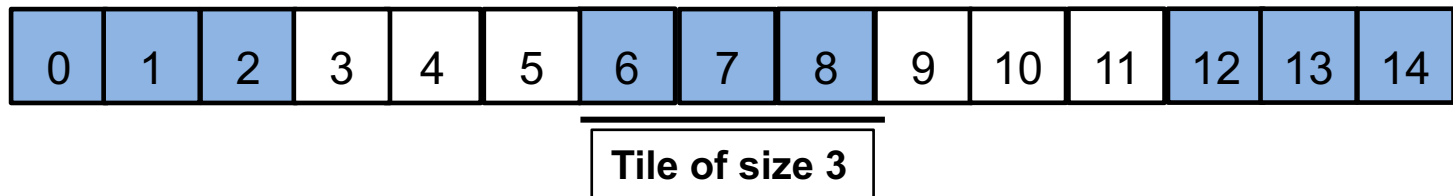
A RAJA::KernelPolicy type is a StatementList.

RAJA provides a variety of RAJA::statement types

- We will describe how to use several of them in this tutorial.
- See the RAJA User Guide for a complete listing of available statement types and what they do

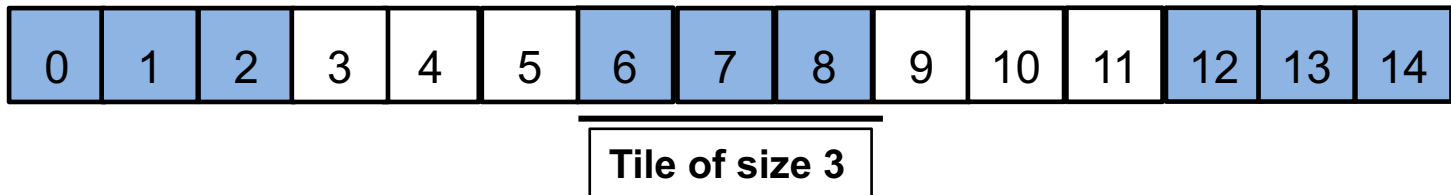
Loop Tiling

Loop tiling enables accessing data in chunks



- Helps ensure data used in a loop stays in a cache until it is reused
- Different levels of memory may be used, tile size is a performance tuning parameter

Loop tiling enables accessing data 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

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

A

$$\begin{pmatrix} a_{00} & a_{10} & \text{ } & \text{ } \\ a_{01} & a_{11} & \text{ } & \text{ } \\ \text{ } & \text{ } & \text{ } & \text{ } \\ \text{ } & \text{ } & \text{ } & \text{ } \end{pmatrix}$$

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

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

A^T

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

C-style matrix transpose without storing local tile

$A^T(c, r) = A(r, c)$, A is $N_r \times N_c$ matrix, A^T is $N_c \times N_r$ 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 over a tile
      for (int tc = 0; tc < TILE_SZ; ++tc) {

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

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

      }
    }
  }
}

```

Note: in general, bounds check is needed to prevent indexing out of bounds.

RAJA tiling policies have analogous structure

```
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 policies have analogous structure

```
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 in tile
                statement::For<0, seq_exec, // cols in tile
```

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

```
        >
```

```
    >
```

```
>
```

```
>
```

```
>;
```

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

Note that global indices are calculated automatically.

Exercise #7: Tiled matrix transpose

- See file: `RAJA/exercises/tutorial_halfday/ex7_tiled-matrix-tranpose.cpp`
 - It contains a C-style sequential implementation of a tiled matrix transpose. It also contains a RAJA tiling policy for a matrix transpose
- *Exercise: Implement the matrix transpose kernel using the RAJA kernel API. Use the provided policy to execute the kernel. Try modifying the policy to use OpenMP threads (do the same for CUDA if you can). The file contains empty code sections indicated by comments for you to fill in and methods you can use to check your work and print results.*

Notes:

- Bounds check not needed. RAJA tiling statements ‘mask’ out-of-bounds indices.
- Global indices are passed into a lambda. No need to compute manually.

Exercise #7 Solution

- Your code should look like the following, where you have provided a kernel using the RAJA kernel API

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

- The file **RAJA/exercises/tutorial_halfday/ex7_tiled-matrix-transpose_solution.cpp** contains complete implementations of the solution to exercise #7.

Local Data

Many algorithms require non-perfectly nested loops to improve performance

- To this point, we have considered perfectly nested loops; i.e., loop nests with no intervening code between loops

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

Recall the matrix
multiplication example

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

Many algorithms require non-perfectly nested loops to improve performance

- To this point, we have considered perfectly nested loops; i.e., loop nests with no intervening code between loops

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

Recall the matrix multiplication example

```
        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?

We use lambda statements to indicate intervening code between loops

```

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

        double dot = 0.0;

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

        C(row, col) = dot;
    }
}

```

```

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

```

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

    RAJA::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;
    }
);

```

Here, all lambdas have the same args, but not all args must be used. RAJA provides other statement types to make this cleaner.

Use the execution policy to compose statements that define the kernel execution pattern

```

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;
            >
        >
    >
>;

```

Nested statements are analogous to nested for-loops and other statements in a C-style loop nest.

Other policies: collapse loops in an OpenMP parallel region

```
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.

Other policies: 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...

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

C-style
'tiled' loop
nest

```
    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.
```

Often, local array usage can improve memory access efficiency.

Note: different parallel strategies have different access requirements for 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 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 inner loop is parallel, tile data should be shared between threads

RAJA's LocalArray type is used to manage these cases in a portable manner

```
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 to create a local array in a kernel.

RAJA's LocalArray type is used to manage these cases in a portable manner

```
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 initialized 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 (#).

RAJA provides several memory policy types for local arrays.

RAJA::cpu_tile_mem – Allocates memory on the CPU stack

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

RAJA::cuda_thread_mem – Allocates memory local to a CUDA thread

The kernel for the matrix transpose with local array uses the RAJA::LocalArray type...

```
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) {
        Aview(col, row) = TileArray(ty, tx);
    }
);
```

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

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

Exercise #8: Matrix transpose with a local array

- See file: **RAJA/exercises/tutorial_halfday/ex8_matrix-transpose-local-array.cpp**
 - It contains a C-style sequential implementation of the matrix transpose which stores tile data in a local array. Additionally, a RAJA sequential kernel version is provided which stores tile data in a RAJA local array.
- *Exercise: Implement the RAJA policy for the OpenMP RAJA kernel variant of matrix transpose which stores tile data in a RAJA local array (do the same for CUDA if you can). The file contains empty code sections indicated by comments for you to fill in and methods you can use to check your work and print results.*

Notes:

- You will need both the local tile index as well as the global index.
- 'ForlCount' statements generate local tile indices passed to lambdas in kernel. 'Param' statements identify index args.
- The 'InitLocalMemory' statement may be used to initialize an array within a RAJA kernel.

Exercise #8 Solution for OpenMP

(See file RAJA/exercises/tutorial_halfday/ex8_matrix-transpose-local-array_solution.cpp)

```
using namespace RAJA;
using RAJA::statement;
using EXEC_POL = KernelPolicy< Tile<1, tile_fixed<TILE_DIM>, omp_parallel_for_exec,
                               Tile<0, tile_fixed<TILE_DIM>, loop_exec,

                               InitLocalMem<cpu_tile_mem, ParamList<2>, // local array is
                                                                           // param tuple
                               ForICount<1, Param<0>, loop_exec, // item 2
                               ForICount<0, Param<1>, loop_exec,
                               Lambda<0>
                               >
                               >,
                               ForICount<0, Param<1>, loop_exec,
                               ForICount<1, Param<0>, loop_exec,
                               Lambda<1>
                               >
                               >
                               > // InitLocalMem
                               > // Tile<0...
                               > // Tile<1...
                               >; // KernelPolicy
```

RAJA support for complex loops

	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

Application considerations

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

- Profile your code to see where performance is most important
 - Do a few kernels dominate runtime? No one kernel takes 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

- 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?
 - 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

- 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

- 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 an appropriate “loop taxonomy” and policy selection for their code.

Performance portability takes effort

- RAJA (like any programming model) is **an enabling technology – not a panacea**
 - 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 to ensure correct parallel execution
 - Recast some patterns as reductions, scans, etc.
 - Move variable declarations to innermost scope to avoid threading issues
 - Virtual functions and C++ STL are problematic for GPU execution

Simpler is almost always better – use simple types and arrays.

Wrap-up

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

Wrap up

	Seq	SIMD	OpenMP (CPU)	OpenMP (target)	CUDA	TBB	HIP
Simple loops	available	available	available	available	available	available	in progress
Reductions	available	in progress	available	in progress	available	available	in progress
Segments & Index sets	available	available	available	available	available	available	in progress
Atomics	available	not available	available	available	available	available	in progress
Scans	available	not available	available	not available	available	available	in progress
Complex Loops	available	available	available	in progress	available	in progress	in progress
Layouts & Views	available	available	available	available	available	available	in progress



= available



= in progress



= not available

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” directory
- The RAJA User Guide
 - Topics we discussed today, 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?



Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.