October 10-12, 2023

# ALCF Hands-on HPC Workshop

# Hands-on Breakout Session

## Kokkos / RAJA

Brian Homerding
Performance Engineer
Argonne Leadership Computing Facility (ALCF)

October 10th, 2023

Argonne NATIONAL LABORATORY

# Kokkos

Argonne NATIONAL LABORATORY

# CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:

```
void axpby(int n, double* z, double alpha, const double* x,
                             double beta,  const double* y) {
  for(int i=0; i<n; i++)
    z[i] = alpha*x[i] + beta*y[i];
}
```

- Kokkos Implementation:

```
void axpby(int n, View<double*> z, double alpha, View<const double*> x,
                                   double beta,  View<const double*> y) {
  parallel_for("AXpBY", n, KOKKOS_LAMBDA ( const int i) {
    z(i) = alpha*x(i) + beta*y(i);
  });
}
```

Parallel Pattern: for loop

String Label: Profiling/Debugging

Execution Policy: do n iterations

Loop Body

Iteration handle: integer index

# CG Solve: The Dot Product

- Simple data parallel loop with reduction: Kokkos::parallel_reduce
- Non trivial in CUDA due to lack of built-in reduction support
- Bandwidth bound
- Serial Implementation:

```
double dot(int n, const double* x, const double* y) {
  double sum = 0.0;
  for(int i=0; i<n; i++)
    sum += x[i]*y[i];
  return sum;
}
```

Parallel Pattern: loop with reduction

Iteration Index + Thread-Local Red. Varible

- Kokkos Implementation:

```
double dot(int n, View<const double*> x, View<const double*> y) {
  double x_dot_y = 0.0;
  parallel_reduce("Dot",n, KOKKOS_LAMBDA (const int i,double& sum) {
    sum += x[i]*y[i];
  }, x_dot_y);
  return x_dot_y;
}
```

# CG Solve: Sparse Matrix Vector Multiply

- Loop over rows
- Dot product of matrix row with a vector
- Example of Non-Tightly nested loops
- Random access on the vector (Texture fetch on GPUs)

Outer loop over matrix rows

Inner dot product row x vector

```
void SPMV(int nrows, const int* A_row_offsets, const int* A_cols,
          const double* A_vals, double* y, const double* x) {
  for(int row=0; row<nrows; ++row) {
    double sum = 0.0;
    int row_start=A_row_offsets[row];
    int row_end=A_row_offsets[row+1];
    for(int i=row_start; i<row_end; ++i) {
      sum += A_vals[i]*x[A_cols[i]];
    }
    y[row] = sum;
  }
}
```

# CG Solve: Sparse Matrix Vector Multiply

```
void SPMV(int nrows, View<const int*> A_row_offsets,
            View<const int*> A_cols, View<const double*> A_vals,
            View<double*> y,
            View<const double*, MemoryTraits< RandomAccess>> x) {

  // Performance heuristic to figure out how many rows to give to a team
  int rows_per_team = get_row_chunking(A_row_offsets);

  parallel_for("SPMV:Hierarchy", TeamPolicy< Schedule< Static > >
       ((nrows+rows_per_team-1)/rows_per_team,AUTO,8),
    KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team) {

    const int first_row = team.league_rank()*rows_per_team;
    const int last_row = first_row+rows_per_team<nrows? first_row+rows_per_team : nrows;

    parallel_for(TeamThreadRange(team,first_row,last_row),[&] (const int row) {
      const int row_start=A_row_offsets[row];
      const int row_length=A_row_offsets[row+1]-row_start;

      double y_row;
      parallel_reduce(ThreadVectorRange(team,row_length),[&] (const int i, double& sum) {
        sum += A_vals(i+row_start)*x(A_cols(i+row_start));
      } , y_row);
      y(row) = y_row;
    });
  });
}
```

Enable Texture Fetch on x

Row x Vector dot product

Distribute rows in workset over team-threads

Team Parallelism over Row Worksets

# Kokkos

```
> git clone https://github.com/kokkos/kokkos.git
> git clone https://github.com/kokkos/kokkos-tutorials.git
> cd kokkos-tutorials/Exercises/01/Begin
```

# Kokkos

## exercise_1_begin.cpp

```
#include <sys/time.h>

// EXERCISE: Include Kokkos_Core.hpp.
//           cmath library unnecessary after.
// #include <Kokkos_Core.hpp>
#include <cmath>

void checkSizes( int &N, int &M, int &S, int &nrepeat );
```

# Kokkos

## exercise_1_begin.cpp

```
// Check sizes.
checkSizes( N, M, S, nrepeat );

// EXERCISE: Initialize Kokkos runtime.
// Include braces to encapsulate code between initialize and finalize calls
// Kokkos::initialize( argc, argv );
// {

// Allocate y, x vectors and Matrix A:
double * const y = new double[ N ];
```

# Kokkos

## exercise_1_begin.cpp

```
// Initialize y vector.
// EXERCISE: Convert outer loop to Kokkos::parallel_for.
for ( int i = 0; i < N; ++i ) {
  y[ i ] = 1;
}

// Initialize x vector.
// EXERCISE: Convert outer loop to Kokkos::parallel_for.
for ( int i = 0; i < M; ++i ) {
  x[ i ] = 1;
}

// Initialize A matrix, note 2D indexing computation.
// EXERCISE: Convert outer loop to Kokkos::parallel_for.
for ( int j = 0; j < N; ++j ) {
  for ( int i = 0; i < M; ++i ) {
    A[ j * M + i ] = 1;
```

# Kokkos

## exercise_1_begin.cpp

```
// Initialize y vector.
Kokkos::parallel_for( "y_init", N, KOKKOS_LAMBDA ( int i ) {
  y[ i ] = 1;
});

// Initialize x vector.
Kokkos::parallel_for( "x_init", M, KOKKOS_LAMBDA ( int i ) {
  x[ i ] = 1;
});

// Initialize A matrix, note 2D indexing computation.
Kokkos::parallel_for( "matrix_init", N, KOKKOS_LAMBDA ( int j ) {
  for ( int i = 0; i < M; ++i ) {
    A[ j * M + i ] = 1;
  }
});
```

# Kokkos

## exercise_1_begin.cpp

```
    // EXERCISE: Convert outer loop to Kokkos::parallel_reduce.
    for ( int j = 0; j < N; ++j ) {
      double temp2 = 0;

      for ( int i = 0; i < M; ++i ) {
        temp2 += A[ j * M + i ] * x[ i ];
      }

      result += y[ j ] * temp2;
    }

```

# Kokkos

## exercise_1_begin.cpp

```
    Kokkos::parallel_reduce( "yAx", N, KOKKOS_LAMBDA ( int j, double &update ) {
      double temp2 = 0;

      for ( int i = 0; i < M; ++i ) {
        temp2 += A[ j * M + i ] * x[ i ];
      }

      update += y[ j ] * temp2;
    }, result );
```

# Kokkos

## exercise_1_begin.cpp

```

  // EXERCISE: finalize Kokkos runtime
  // }
  // Kokkos::finalize();

  return 0;
}
```



Argonne NATIONAL LABORATORY

# Kokkos

## Build and Environment

### POLARIS

```
> module swap PrgEnv-nvhpc/8.3.3 PrgEnv-gnu
> module load nvhpc-mixed
> cd /path/to/kokkos-tutorials/02/Begin



> make KOKKOS_PATH=/path/to/kokkos KOKKOS_DEVICES="Cuda" KOKKOS_ARCH="Ampere80"
```

# RAJA

# RAJA loop execution has four core concepts

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

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

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

# Reduction is a common and important parallel pattern

dot product: $dot = \sum_{i=0}^{N-1} a_i\, b_i$, where a and b are vectors, dot is a scalar

C-style

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

# A RAJA reduction object hides the complexity of a parallel reduction operation

C-style

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

RAJA

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

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

# Elements of RAJA reductions…

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

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

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

- A **reduction type** requires:
  - A reduction policy
  - A reduction value type
  - An initial value

# Elements of RAJA reductions...

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

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

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

Note that you cannot access the reduced value inside a kernel.

- A reduction type requires:
  - A reduction policy
  - A reduction value type
  - An initial value
- **Updating reduction value is what you expect (+=, min, max)**

# Elements of RAJA reductions…

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

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

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

- A reduction type requires:
  - A reduction policy
  - A reduction value type
  - An initial value
- Updating reduction value is what you expect (+=, min, max)
- **After loop runs, get reduced value via ‘get’ method**

# The reduction policy must be compatible with the loop execution policy

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

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

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

An OpenMP execution policy requires an OpenMP reduction policy, similarly for CUDA, etc.

# RAJA provides reduction policies for all supported programming model back-ends

```
RAJA::ReduceSum< REDUCE_POLICY, int > sum(0);
```

```
RAJA::seq_reduce;

RAJA::omp_reduce;

RAJA::cuda_reduce;

RAJA::tbb_reduce;

RAJA::omp_target_reduce;
```

A sample of RAJA reduction policy types.

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

# RAJA supports five common reductions types

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

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

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

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

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

Initial “loc” values

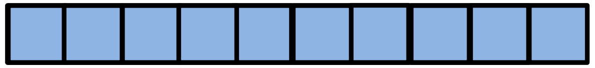
“Loc” reductions give a loop index where reduced value was found.

# A RAJA “Segment” defines a set of loop iterates

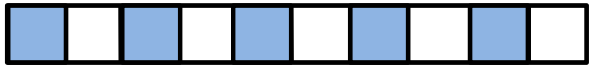
**Iteration spaces**

- A **Segment** is a set of loop indices to run for a kernel

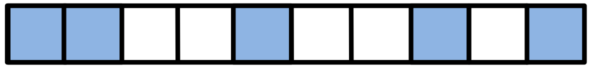
**Contiguous range** [beg, end)



**Strided range** [beg, end, stride)



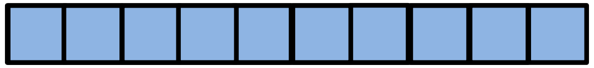
**List of indices** (indirection)



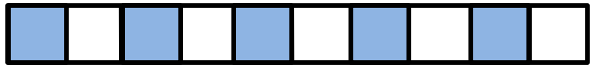
# Loop iteration spaces are defined by Segments

- A Segment is a set of loop indices to run for a kernel

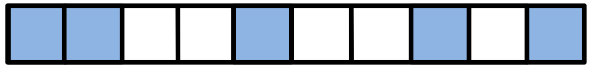
**Contiguous range** [beg, end)



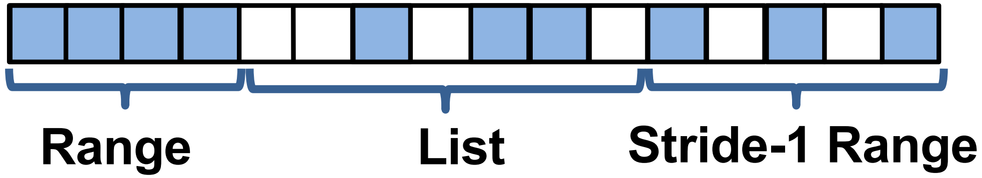
**Strided range** [beg, end, stride)



**List of indices** (indirection)



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




You can run all Segments in an IndexSet in one call to a 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

Iteration spaces

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

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

Runs loop indices: 0, 2, 4, …

# Segments also allow negative indices and strides

Iteration spaces

```
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

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

Nested loops

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

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

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

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

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

# Each loop level has an iteration space and loop variable

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

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

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

# Each loop level has an execution policy

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

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

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

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

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

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

'For' statements are swapped.

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

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

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

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

Nested loops

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

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

```
Lambda<0>
```

```
CudaSyncThreads
```

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

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

A RAJA::KernelPolicy type is a StatementList.

# RAJA

```
> git clone --recursive https://github.com/llnl/raja.git
> cd raja/exercises/tutorial_halfday
```

# RAJA

## ex1_vector-addition.cpp

```
/// EXERCISE: Implement the vector addition kernel using a RAJA::forall
///           method and RAJA::seq_exec execution policy type.
///
/// NOTE: We've done this one for you to help you get started...
///

using EXEC_POL1 = RAJA::seq_exec;

RAJA::forall< EXEC_POL1 >(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});

checkResult(c, c_ref, N);
```

# RAJA

## ex1_vector-addition.cpp

```
std::cout << "\n Running RAJA SIMD vector addition...\n";

///
/// TODO...
///
/// EXERCISE: Implement the vector addition kernel using a RAJA::forall
///           method and RAJA::simd_exec execution policy type.
///

checkResult(c, c_ref, N);
```

# RAJA

## ex1_vector–addition.cpp

```
std::cout << "\n Running RAJA SIMD vector addition...\n";

using EXEC_POL2 = RAJA::simd_exec;

RAJA::forall< EXEC_POL2 >(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});

checkResult(c, c_ref, N);
```

# RAJA

## ex1_vector-addition.cpp

```
std::cout << "\n Running RAJA OpenMP multithreaded vector addition...\n";

///
/// TODO...
///
/// EXERCISE: Implement the vector addition kernel using a RAJA::forall
///           method and RAJA::omp_parallel_for_exec execution policy type.
///

checkResult(c, c_ref, N);
```

# RAJA

## ex1_vector-addition.cpp

```
std::cout << "\n Running RAJA OpenMP multithreaded vector addition...\n";

using EXEC_POL4 = RAJA::omp_parallel_for_exec;

RAJA::forall< EXEC_POL4 >(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});

checkResult(c, c_ref, N);
```

# RAJA

## ex1_vector–addition.cpp

```
std::cout << "\n Running RAJA CUDA vector addition...\n";

///
/// TODO...
///
/// EXERCISE: Implement the vector addition kernel using a RAJA::forall
///           method and RAJA::cuda_exec execution policy type.
///

checkResult(c, c_ref, N);
```

# RAJA

## ex1_vector-addition.cpp

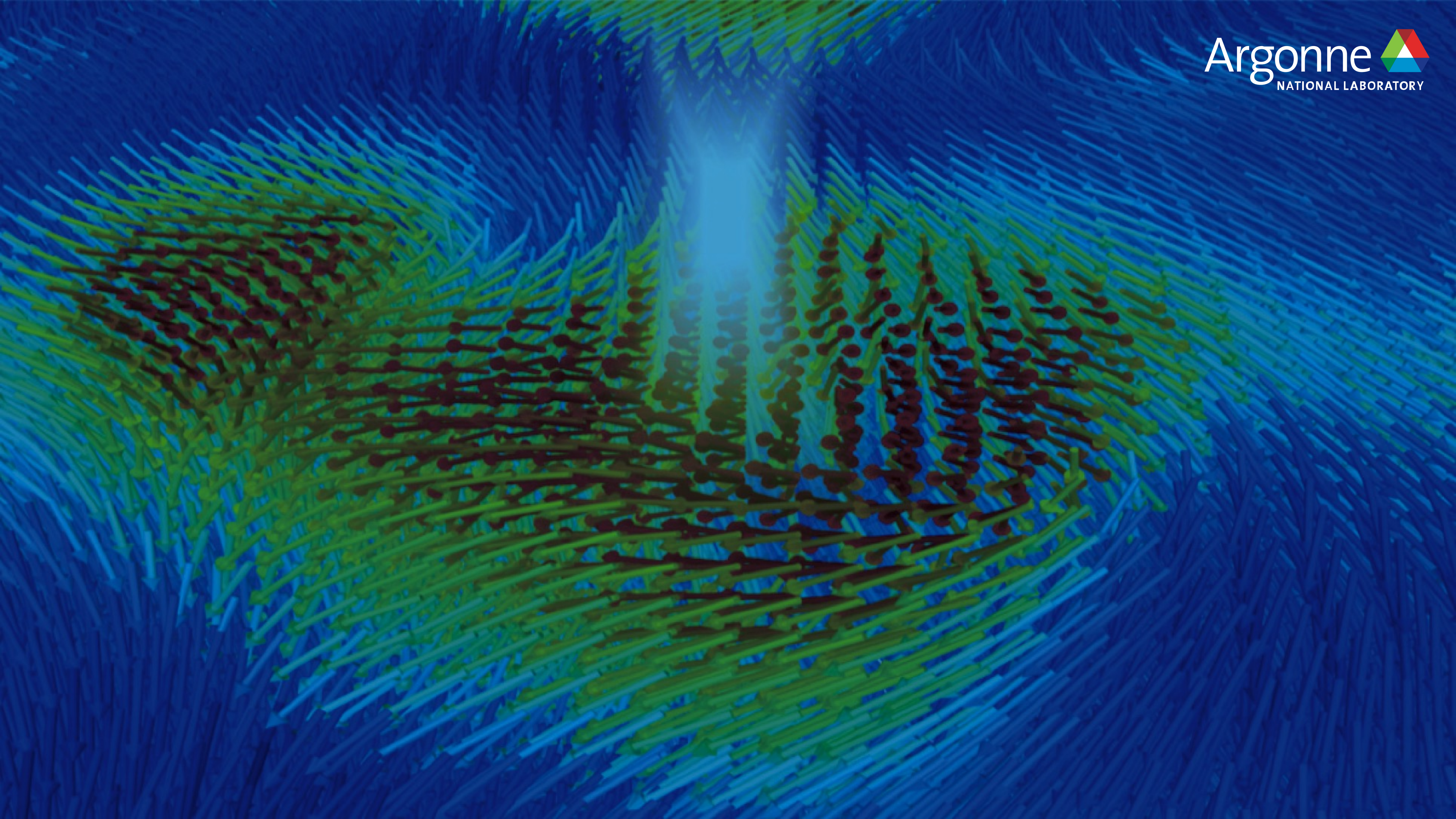
```
#if defined(RAJA_ENABLE_CUDA)
const int CUDA_BLOCK_SIZE = 256;
#endif
{…}
  std::cout << "\n Running RAJA CUDA vector addition...\n";

  using EXEC_POL5 = RAJA::cuda_exec<CUDA_BLOCK_SIZE>;

  RAJA::forall< EXEC_POL5 >(RAJA::RangeSegment(0, N),
                            [=] RAJA_DEVICE (int i) {
    c[i] = a[i] + b[i];
  });

  checkResult(c, c_ref, N);
```

# RAJA

## Build and Environment

### POLARIS

```
> module swap PrgEnv-nvhpc/8.3.3 PrgEnv-gnu
> module load nvhpc-mixed cmake
> cd /path/to/raja
> mkdir build
> cd build
> cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=g++ -DCMAKE_C_COMPILER=gcc
       -DCUDA_COMMON_OPT_FLAGS="-restrict -arch sm_80 --expt-extended-lambda"
        -C ../host-configs/ubuntu-builds/nvcc_gcc_X.cmake
       -DENABLE_CUDA=On -DRAJA_ENABLE_EXAMPLES=On ..
> Make -j 16
```


Argonne
NATIONAL LABORATORY