The RAJA Encapsulation Model for Architecture Portability

Berkeley C++ Summit

David Beckingsale

October 17, 2016

LLNL-PRES-705349

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

Lawrence Livermore National Laboratory

Production applications contains 10,000s of C-style for loops

```
double* x ; double* y ;
double a, tsum = 0, tmin = MYMAX;

for ( int i = begin; i < end; ++i )  {
     y[i] += a * x[i] ;
     tsum += y[i] ;
     if ( y[i] < tmin ) tmin = y[i];
}
```

RAJA provides an abstraction that encapsulates loop execution details

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

Reduction types provide portability across programming models

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

Traversal templates specialized on execution policies control loop scheduling & execution

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

IndexSets allow iteration space partitioning, ordering, and dependencies

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

Loop body written as lambda function

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
    y[i] += a * x[i] ;
    tsum += y[i];
    tmin.min( y[i] );
});
```

Loop body remains largely untouched

```
for ( int i = begin; i < end; ++i )  {
      y[i] += a * x[i] ;
      tsum += y[i] ;
      if ( y[i] < tmin ) tmin = y[i];
}
```

```
RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
      y[i] += a * x[i] ;
      tsum += y[i];
      tmin.min( y[i] );
});
```

RAJA aims to enable performance portability with minimal disruption to the user's programming style

Balance Performance:

- Augment compiler's ability to optimize existing C++ code
  - Enable work-arounds when performance is not what's expected
- Expose various forms of fine-grained (on-node) parallelism

And Productivity:

- Single-source kernels
  - Don't bind an application to a particular PM technology
  - Best choice for a given platform or algorithm may not be clear
- Clear separation of responsibilities
  - **RAJA:** Execute loop iterations, handle hardware & PM details
  - **App:** Select iteration patterns and execution policies with RAJA API

App developers add parallelism to "serial" code using RAJA.
Preserve development dynamics and advantages of MPI heritage

RAJA Concepts

Execution policies control loop scheduling & execution

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

Sequential execution

```
forall(seq_exec, int begin, int end, LB body)
{
   for(int i=begin; i < end; i++) {
     body(i);
   }
}
```

- Sequential execution is like a regular for loop
- Allow users to port kernels that aren't thread safe

OpenMP execution policy

```
forall(omp_exec, int begin, int end, LB body)
{
#pragma omp parallel for
  for(int i=begin; i < end; i++) {
    body(i);
  }
}
```

- OpenMP parallel execution starts an OpenMP parallel region around the loop

CUDA execution policy

```
forall(cuda_exec<BLOCK>, int begin, int end, LB body)
{
  int len = end - begin;
  size_t gridSize = (len + BLOCK - 1)/ BLOCK;

  forall_cuda_kernel<<<gridSize,BLOCK>>>
    (loop_body, begin, len);
}
```

- CUDA execution dispatches the loop body iterations to the GPU
- We can map each iteration of the space to a different CUDA thread
- Launch a grid of threads big enough to cover the entire space
- Template parameter controls block size

CUDA execution policy – device kernel

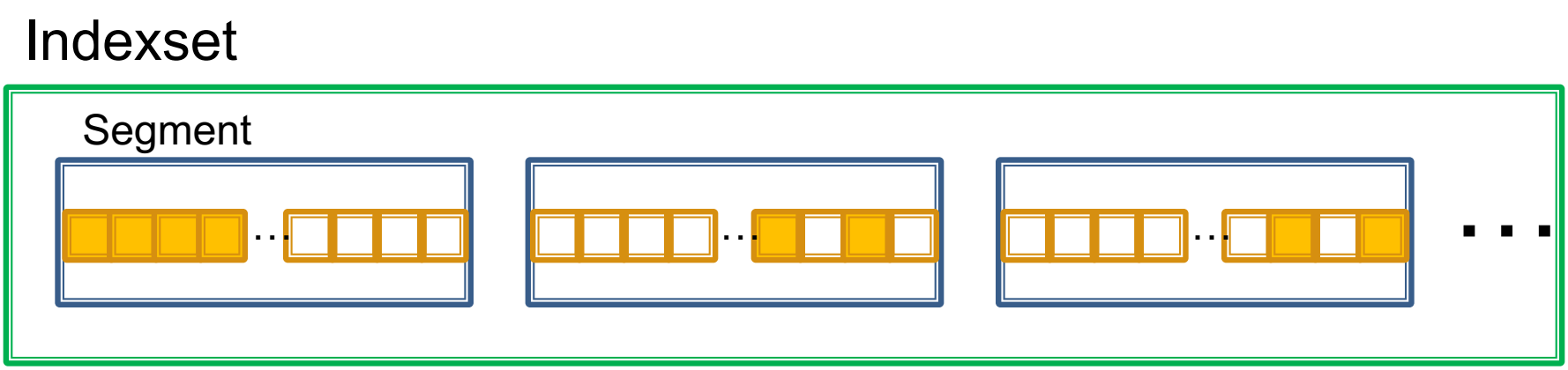
```
forall_cuda_kernel(LB loop_body, int begin, int len) {
  Index_type ii = blockDim.x * blockIdx.x + threadIdx.x;
  if (ii < len) {
    loop_body(begin + ii);
  }
}
```

- The CUDA kernel executes the loop body, *if* the index is in range

IndexSets allow iteration space partitioning, ordering, and dependencies

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

An *IndexSet* partitions an index space into *Segments* that cover the space

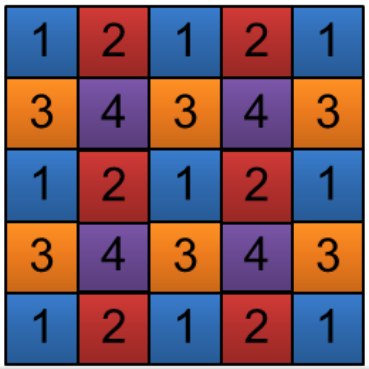


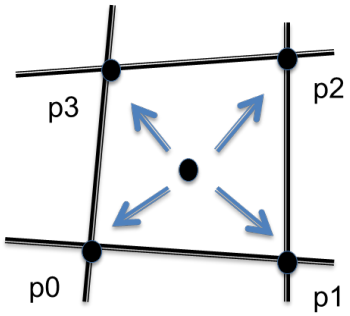
- An Indexset is a container for Segments
- A Segment is a container for iterates
- Segments can be executed in serial, parallel, or according to dependency relationships
- Iterates can be executed in serial or parallel

RAJA currently supports up to two levels of parallelism

RAJA IndexSets simplify thread-safe refactoring of code with data races

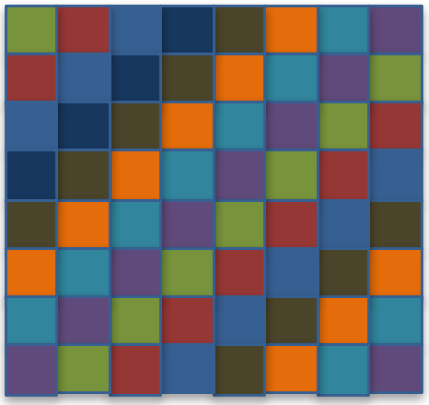
```
forall<colorset>(elemSet, [=] (int elem) {
  // Get vertex ids
  int p0 = elemToNodeMap[elem][0];
  int p1 = elemToNodeMap[elem][1];
  int p2 = elemToNodeMap[elem][2];
  int p3 = elemToNodeMap[elem][3];

  // Accumulate volume to vertices
  double volFrac = elemVol[elem]/4.0 ;
  nodeVol[p0] += volFrac ;
  nodeVol[p1] += volFrac ;
  nodeVol[p2] += volFrac ;
  nodeVol[p3] += volFrac ;
}) ;
```





- **Example:** this code accumulates element-centered volume to vertices.
- Iterations can be partitioned into colored sets of independent work (Segments) that can be executed in parallel.
  - Reordering iterations avoids a data race during the vertex sum.
- Without reordering, requires one of:
  - Contention-heavy fine-grained synchronization (atomics / critical sections)
  - Temporary arrays for accumulating sums
- RAJA reordering allows use of coarse-grained synchronization.
  - Light memory contention
  - Still easy to read!

IndexSets naturally describe deterministic transport sweep dependencies



Hyper-plane IndexSet
- 15 Hyper-planes
- 1 Segment per hyper-plane

```
RAJA::forall<sweep_policy>(hplane_is,
 [=](int i, int j, int k){
    // Do Spatial Operator
 });
```

Choose with **`sweep_policy`** `typedef`

OpenMP
- Iterate over segments sequentially
- Iterate over cells in a segment in parallel

CUDA
- Iterate over segments sequentially
- Launch CUDA kernel for each segment

Reduction types provide portability across programming models

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

Reduction logic for each programming model hidden inside class, specialized on reduction policy

```
SumReduction<omp_reduce, T> operator+=(T val) const
{
  int tid = omp_get_thread_num();
  m_blockdata[tid * s_block_offset] += val;
  return *this;
}
```

- The reduction policy must match the execution policy, but we are exploring ways to unify this

forallN is a RAJA extension for nested-loops

forallN() abstracts a N-nested loop

- Execution policy for each index
- Loop interchange permutation

Index types (ordered)
(Optional: defaults to int)

```
forallN<SweepPolicy<nest_type>, IDirection, IGroup, IZoneIdx>(
 domain->indexRange<IDirection>(sdom_id),
 domain->indexRange<IGroup>(sdom_id),
 extent.indexset_sweep,
 [=](IDirection d, IGroup g, IZoneIdx zone_idx){
   // Do some work
   psi(d,g,z) *= sigma(g);
 });
```

IndexSet's for each Index (ordered)

Views abstract index calculations and provide type safety

Loop body with type-safe indices

Extensions provide nested loop concepts and code correctness features

RAJA::forallN is equivalent to nested RAJA::foralls with loop transformations

RAJA::forallN

```
forallN<MyPolicy>(is_i, is_j, is_k, is_l,
  [=](int i, int j, int k, int l){
  fcn(i,j,k,l);
});
```

Simple Policy:

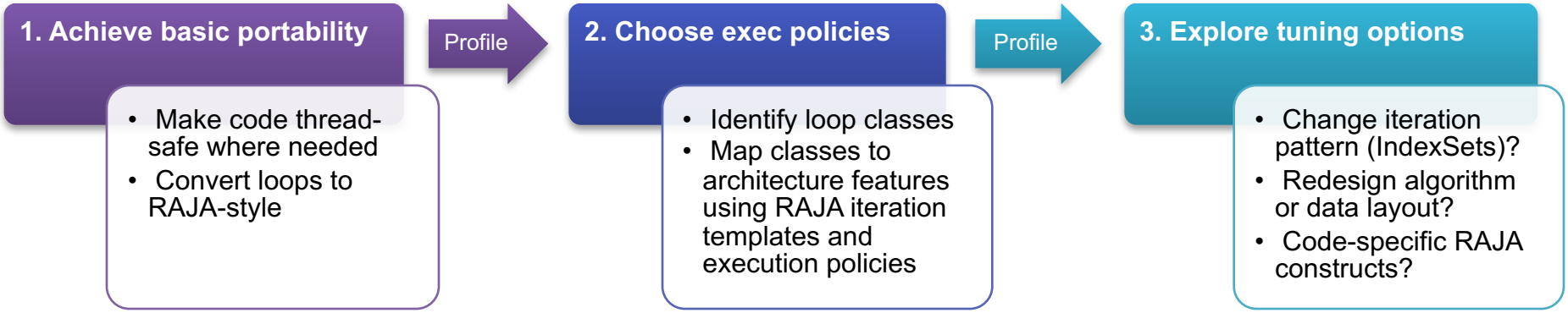
```
forall<pol_i>(is_i, [=](int i){
  forall<pol_j>(is_j, [=](int j){
    forall<pol_k>(is_k, [=](int k){
      forall<pol_l>(is_l, [=](int l){
        fcn(i,j,k,l);
      });
    });
  });
});
```

Loop Interchange Policy:

```
forall<pol_i>(is_i, [=](int i){
  forall<pol_k>(is_k, [=](int k){
    forall<pol_j>(is_j, [=](int j){
      forall<pol_l>(is_l, [=](int l){
        fcn(i,j,k,l);
      });
    });
  });
});
```

RAJA Application Performance

RAJA enables systematic tuning of large production codes



- **Typical LLNL multi-physics code contains O(10K) loops, but O(10) loop *patterns***
  - RAJA allows loop *patterns* to be tuned together
  - Teams typically develop app-specific idioms for their particular loop patterns
- **RAJA allows for easy incremental adoption:**
  - Can be used selectively and incrementally in production codes
  - Existing loop bodies remain essentially unchanged, usually untouched
- **Once RAJA loops are added, developers set exec policies for patterns**
  - Consider data motion, compute intensity, branch intensity, available parallelism
  - In RAJA, it is easy to change and/or revert an execution policy if one does not work.

RAJA core abstractions can combined with application-specific implementations

```
// Kernel 1
for (int i=begin; i<end; ++i) {
  Loop body 1 (stride-1)
}
…
// Kernel 2
for (int i=0; i<len; ++i) {
  Loop body 2 (indirection)
}
…
// Kernel 3
…
// Kernel 4
…
```

Parameterized RAJA loops

```
// Kernel 1 : "stream"  low FLOP/bandwidth
RAJA::forall<stream> ( begin, end, [=] (int i)  {
     Loop body 1
});

// …
// Kernel 2 : "work"  high FLOP/bandwidth
RAJA::forall<work> ( iset, [=] (int i)  {
    Loop body 2 (iset = index "ranges" & "lists")
});
```

"RAJA-fied" app code

Customized RAJA

Execution policies (app defined)
Arch A : `stream = seq, work = omp`
Arch B : `stream = omp, work = gpu`

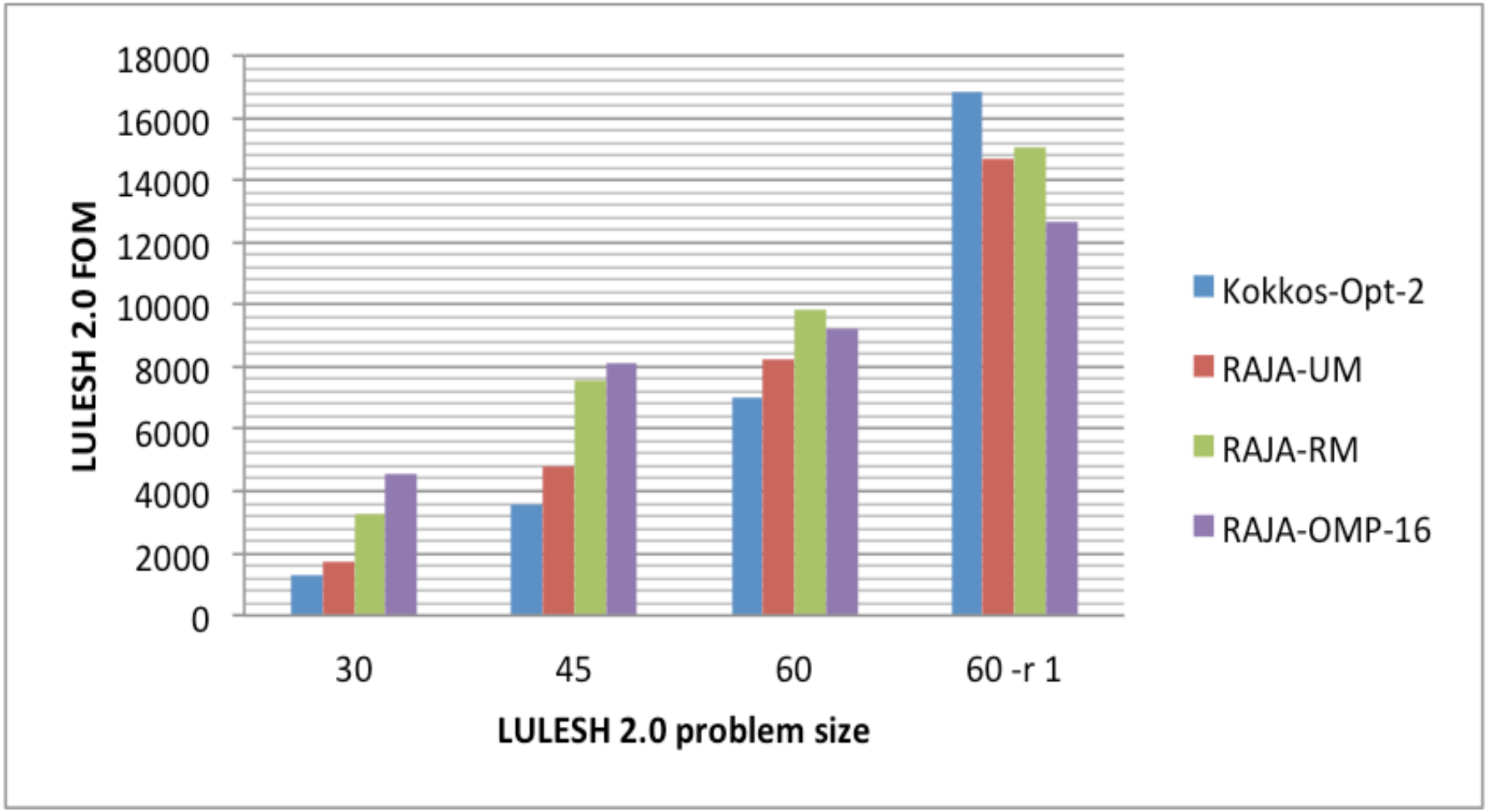
```
// Kernel 3 : RAJA IndexSet w/ custom traversal
app::forall<app_policy> ( iset, [=] (…)  {
     Loop body 3
});
```

Architecture-tailored implementations

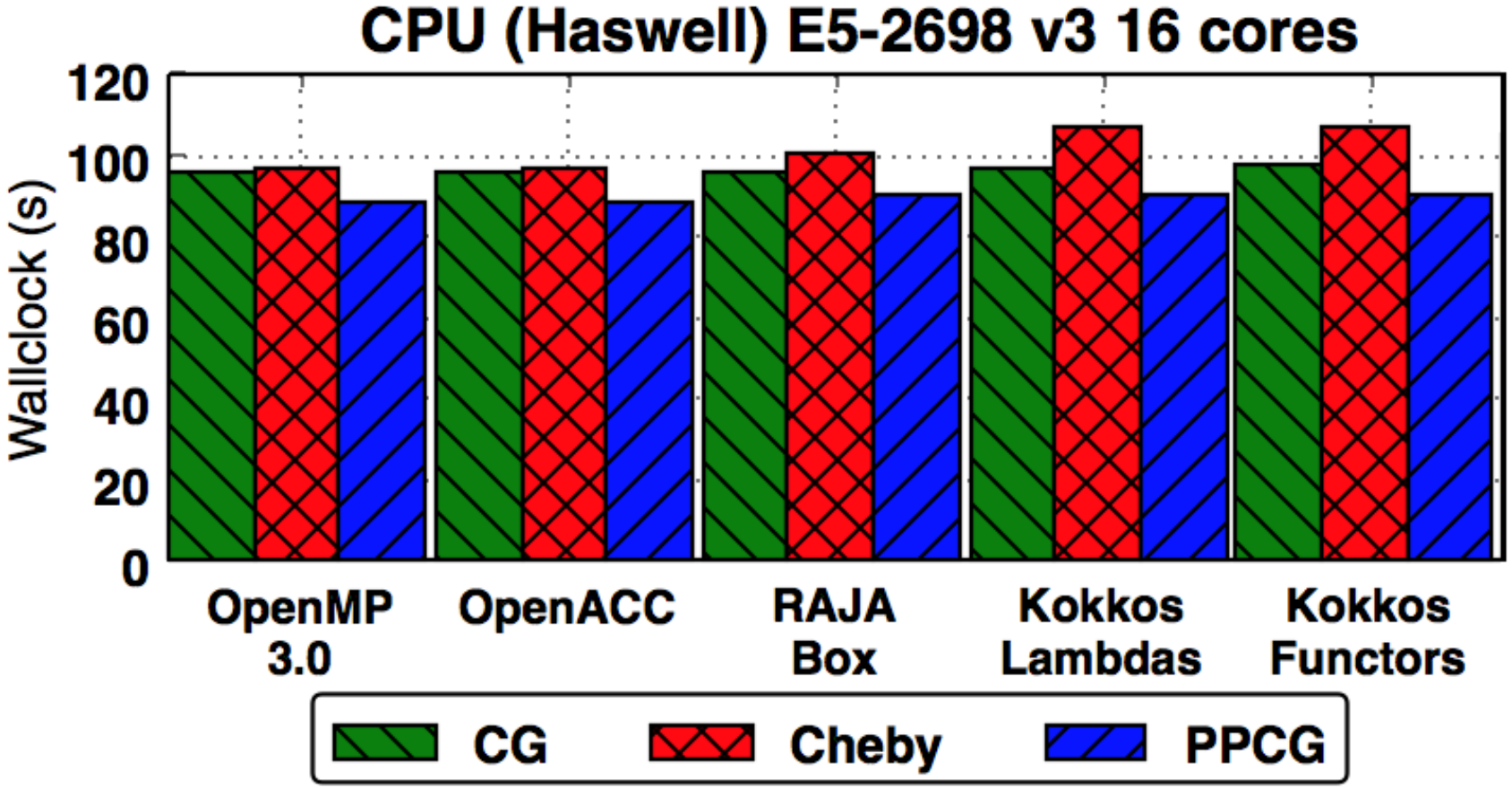
- Kernel 4 - Arch A
- Kernel 4 - Arch B

RAJA enables parameterization of most kernels.
Others may need customization or more severe disruption for desired performance.

RAJA K80 GPU performance for LULESH v2.0 is comparable to Kokkos and to CPU performance



University of Bristol reports TeaLeaf RAJA has the simplest API and performance similar to other PMs



Results courtesy of Matt Martineau

LLNL ASC app teams evaluated and quickly embraced RAJA adoption for Sierra readiness

- Expected benefits (subsequently validated)
  - Incremental adoption, low overhead for initial entry
  - The `forall` idiom matches application loop patterns and insulates hardware and programming model details from users
  - Performance improvements by RAJA team easily integrated with simple library updates
  - Centralized execution policies ease targeting different architectures
  - Enables work-arounds for potential show stoppers (compilers, runtimes)
  - No negative impact on performance for current platforms

- Issues
  - Adopting RAJA and C++ lambda still requires changing some code idioms
  - Lack of a memory model for placement and motion between spaces (actually a RAJA design point)
  - BG/Q XLC compiler does not support C++11
  - RAJA is not a solution for Fortran codes

RAJA was the enabling technology that helped production app teams make rapid progress toward targeting new architectures

LLNL production ASC codes face similar constraints, concerns and experiences

- Common constraints (which RAJA satisfies)
  - Cannot be rewritten for each new programming model or architecture
  - Cannot contain a lot of conditionally compiled code
  - Must always work on all supported platforms
  - True portability requires 'least common denominator' approach – but, encapsulation (MPI, memory allocation) is a tried-and-true approach
- Common concerns:
  - "Best choice" programming model is unclear or platform specific
  - Memory space heterogeneity & NUMA are unsolved challenges
  - How well future hw & sw will perform is a big unknown (e.g., UM)
- Common prep work and experiences:
  - Thread safety & other changes for GPU needed independent of PM
  - Loop analysis and characterization, define code-specific style/idiom using RAJA
  - Change some idioms to enable conversion of loop bodies to lambda functions (reduction types, recast some patterns as reductions, loop control logic)
  - RAJA & lambda correctness often helped by compiler (error messages)
  - Productivity hindered by immature and deficient tools (nvcc device lambda support, UM issues, OpenMP 4.5 stability, debuggers, profilers, etc.)

ALE3D chose to encapsulate RAJA under a custom macro layer and develop CHAI for data transfers

Typical ALE3D code pre-RAJA/CHAI

```
View* elems = problem->view("elems");
View* faces = problem->view("faces");

int nFaces = faces->length();
Relation& faceToelem = *face->relation("faceToElem");

real8 *       F0 = faces->fieldReal("F0");
real8 const * mass = elems->fieldReal("mass");

for ( int i = 0; i < nFaces; ++i )  {
    F0[i] = f( mass[ faceToElem(i) ] );
}
```

Transformed code

```
typedef ManagedArray<double> double_ptr;
typedef ManagedArray<double const> double_const_ptr;

// ...

double_ptr       F0 = faces->fieldReal("F0");
double_const_ptr mass = elems->fieldReal("mass");

FACE_LOOP_BEGIN( i, 0, nFaces )  {
  F0[i] = f( mass[ faceToElem(i) ] );
} FACE_LOOP_END
```

Data transparently copied as needed via C++ copy constructors during lambda capture / kernel launch

- ManagedArray object know *what* data to copy
- CHAI's resource manager knows *whether* to copy
- RAJA provides context for *where* to copy data

Ares chose to encapsulate RAJA under a custom template layer and use UM for data transfers

Typical Ares code pre-RAJA

```
for ( int ir = 1; ir <= nreg; ++ir ) {
  int  rlencln = domain->rlencln[ ir ];
  int* ndx     = domain->rndx[ ir ];

  for ( int i = 0; i < rlencln; ++i ) {
     int zone = ndx[ i ];
     // compute using zone as array index
  }
}
```

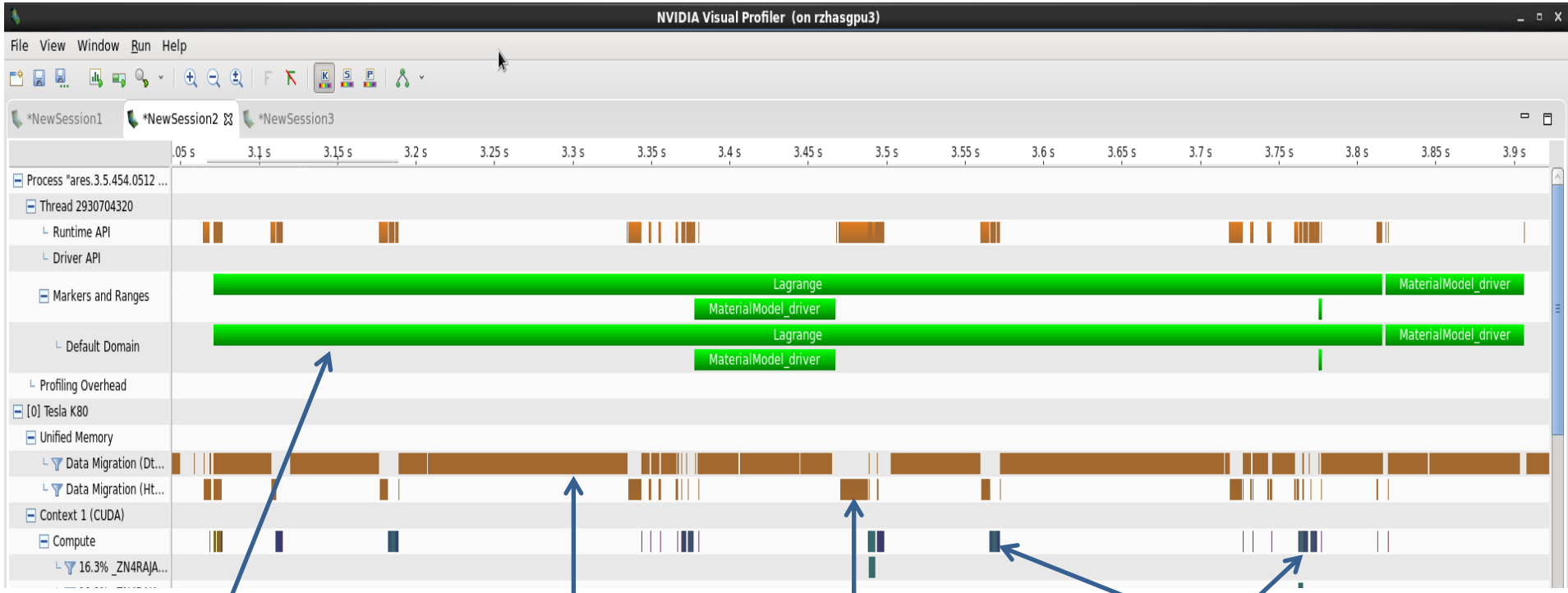
Transformed code

```
for ( int ir = 1; ir <= nreg; ++ir ) {

  for_all_clean_zones< policy::cuda >(
     domain, ir, ARES_LAMBDA(int zone) {
     // compute using zone as array index
  )};
}
```

Data transparently copied as needed via UM paging

Custom loop traversal template passes appropriate RAJA IndexSet on domain to RAJA

Runtime of initial Ares port to GPU was dominated by data motion



Sections explicitly marked in the code

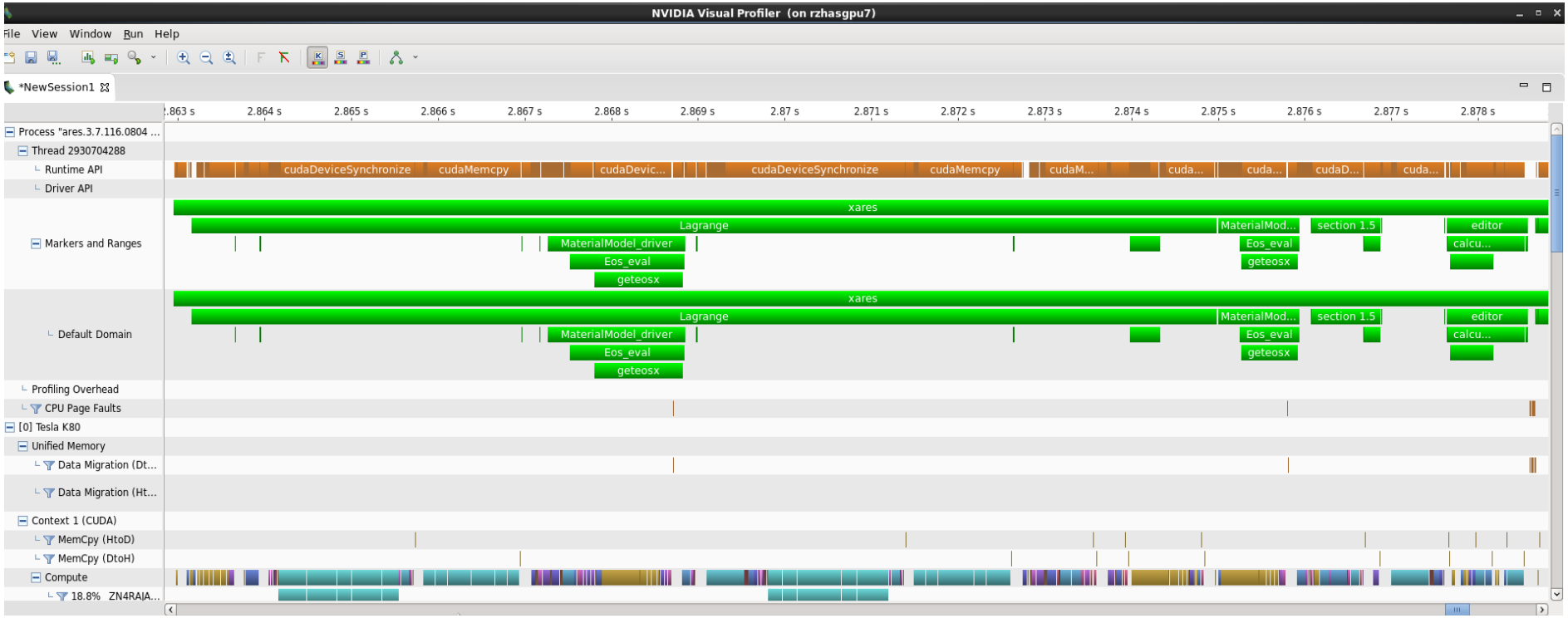
Data motion (device-to-host)

Data motion (host-to-device)

GPU kernels

Profile shows one time step in Lagrange hydro algorithm

Systematically reducing memory motion improved performance dramatically



Speedup over serial = 10.5
Speedup over 16 MPI tasks = 1.14

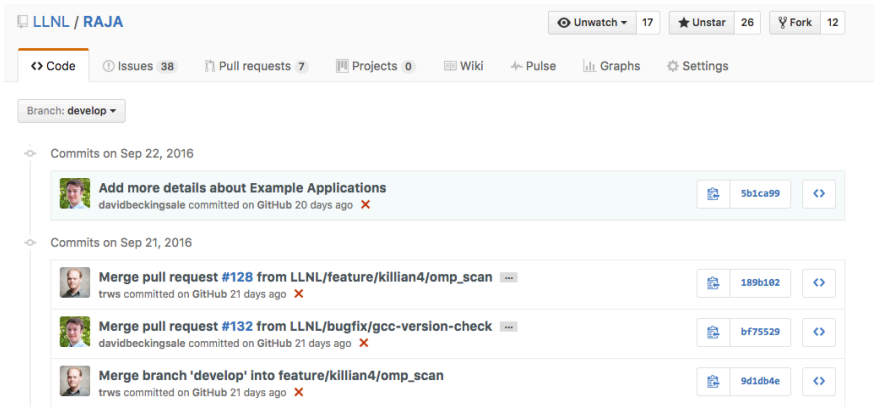
Profile shows one time step in Lagrange hydro algorithm

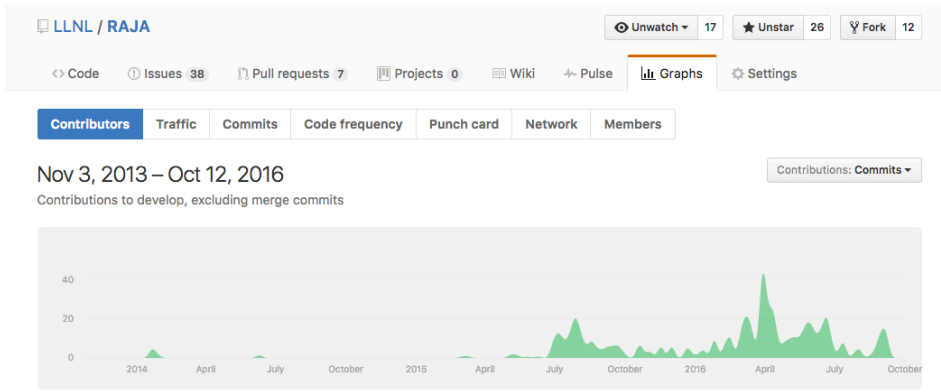
RAJA enables a single legacy source code base to run with multiple forms of parallelism

- CPU-GPU portability with existing programming models + standard C++11
  - Often does not require code restructuring or multiple code versions
  - Many reduction operations do not need be restructured for parallel execution
- RAJA makes tuning more systematic
  - Tune loop patterns, not individual loops
- IndexSets provide significant flexibility
  - Code specializations generated and optimized at compile-time, applied to data at runtime
  - Dependencies between iteration subsets can be parallelized without critical sections
- RAJA is being integrated with LLNL production apps for Trinity & Sierra:
  - Integrating & contributing to RAJA: ARES, ALE3D, ARDRA, NIF VBL
  - Exploring RAJA: Hypre, VisIt, Hydra, MARBL
- RAJA has already spawned a number of extension projects:
  - CHAI data movement abstraction layer
  - Nested loops & loop interchange
  - RAJA gives tool developers a ready handles for auto-tuning

RAJA has been released and is under active development on GitHub

- https://github.com/LLNL/RAJA
- LLNL commitment to open-source releases is gathering momentum
- Helps facilitate collaboration with external users, as well as vendors
- RAJA proxy applications are available at: https://github.com/LLNL/RAJA-examples
- We are working on developing a performance suite to quantify and track how well compilers are optimizing our code over time





Acknowledgements

- Rich Hornung and the rest of the RAJA contributors:
  - Adam Kunen, David Poliakoff, Tom Scogland, Holger Jones, Will Killian, Jeff Keasler
- Brian Ryujin, Jason Burmark and George Zagaris: Ares work
- Peter Robinson: ALE3D work


Lawrence Livermore
National Laboratory